RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs

Davide Corradini*, Amedeo Zampieri†, Michele Pasqua‡ and Mariano Ceccato§
Department of Computer Science
University of Verona – Verona, Italy
Email: *davide.corradini@univr.it, †amedeo.zampieri@studenti.univr.it, ‡michele.pasqua@univr.it, §mariano.ceccato@univr.it

***Abstract*—Over the past few years, more and more novel black-box testing approaches for RESTful APIs have been proposed. In order to assess their effectiveness, such testing strategies had to be implemented as a prototype tool and validated on empirical data. However, developing a testing tool is a time-consuming task, and reimplementing from scratch the same common basic features represents a waste of resources that causes a remarkable overhead in the "time to market" of research results.**

In this paper, we present RestTestGen, an extensible framework for implementing new automated black-box testing strategies for RESTful APIs. The framework provides a collection of commonly used components, such as a robust OpenAPI specification parser, dictionaries, input value generators, mutation operators, oracles, and others. Many of the provided components are customizable and extensible, enabling researchers and practitioners to quickly prototype, deploy, and evaluate their novel ideas. Additionally, the framework facilitates the development of novel black-box testing strategies by guiding researchers, by means of abstract components that explicitly identify those parts of the framework requiring a concrete implementation.

As an adoption example, we show how we can implement nominal and error black-box testing strategies for RESTful APIs, by reusing primitives and features provided by the framework, and by concretely extending very few abstract components.

RestTestGen is open-source, actively maintained, and available on GitHub at https://github.com/SeUniVr/RestTestGen

***Keywords*-REST API, Test case generation, Black-box testing**

I. Introduction

An API that respects the REpresentational State Transfer (REST) architectural style [1] is termed a *RESTful* API (or REST API for short). REST APIs are becoming a de-facto industrial standard to interconnect different computer systems, for instance, when exchanging data with the cloud [2], when connecting smartphone apps to their corresponding server [3], for identity provisioning [4], and for banks inter-operation [5].

Automated testing of RESTful APIs is an emerging topic in the software engineering research community, and several approaches have been proposed to this aim. Some of them are *white-box*, and rely on the availability of source code to perform static analysis, or to instrument it to collect execution traces and metric values (e.g., EvoMaster [6], that generate test cases for Java-based REST APIs by means of evolutionary algorithms).

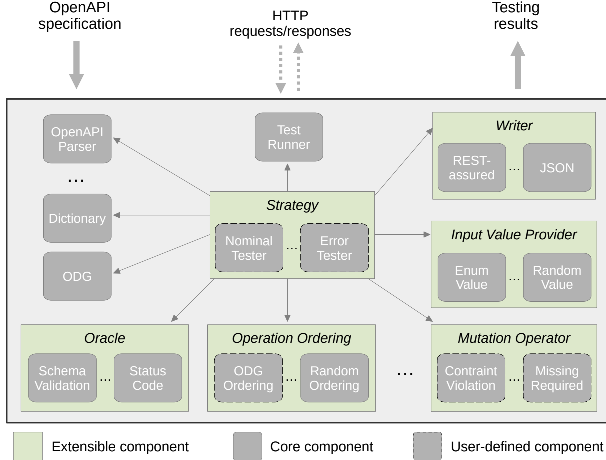
Since the source code of REST APIs is very often not available, *black-box* approaches, that do not require any source code, are usually preferable. Here we can find *fuzzers* (e.g., APIFuzzer [7], FuzzLightyear [8], TnTFuzzer [9]), that generate new tests starting from previously recorded API traffic: they fuzz and replay new traffic in order to find bugs. Other tools, in addition, exploit the OpenAPI specification of the service under test (e.g., RESTler [10], QuickREST [11], bBOXRT [12]), in order to generate more effective test cases.

Segura et al. [13] propose a black-box approach where the oracle is based on metamorphic relations among requests and responses. Martin-Lopez et al. [14] present a domain-specific language (IDL) for the specification of dependencies among input parameters in web services. Then, they translate an IDL document into a constraint satisfaction problem (CSP), enabling the automated analysis of IDL specifications using standard CSP-based reasoning operations. Huayao et al. [15] propose an approach that relies on *combinatorial testing* to generate parameter input values, and that leverages *natural language processing* to implement a pattern-based approach for extracting inter-parameter constraints.

Developing all these testing tools or research prototypes required to re-implement the same common basic features over and over again, causing time and resources to be wasted in engineering-intensive tasks, therefore reducing those available for research-intensive tasks. To the best of our knowledge there is no *framework* that facilitates researchers and practitioners in defining new testing strategies for REST APIs by simplifying tool development.

In this work, we present RestTestGen, a modular and extensible framework for implementing new automated black-box testing strategies for RESTful APIs. The architecture of this tool is shown in Figure 1. It provides some basic features commonly used in REST API testing that works out-of-the-box, namely the *core components*, that will be presented in Section II. Moreover, in order to deliver novel and customized testing strategies, the framework also provides some *extensible components* that researchers or practitioners can implement and extend. They will be described in Section III. Core and extensible components represents the basic blocks to assemble to conceive new testing approaches. Complete and working research testing strategies [16] are already available in the framework (Section IV), as examples of how to instantiate and assemble all the needed components. We will show how to run them to automatically generate tests on a REST API case study in Section V.

Fig. 1. A high-level view of the architecture of RestTestGen.



II. Core Components

Core components are a collection of ready-to-use classes, that a researcher may directly integrate in its testing strategy.

The starting point for REST API black-box testing is their OpenAPI specifications. Very often, specifications contain inconsistencies or wrong content, such that the official swagger.io parser [17] might be ineffective [18]. A robust **OpenAPI Specification Parser** is available in RestTestGen, that is more tolerant with unexpected or inconsistent content. Properties not belonging to the OpenAPI grammar are silently ignored and inconsistencies in the specification are fixed, when possible. For instance, type conversion is attempted in case of type mismatch, e.g., an integer parameter with a string value as example or enum values (i.e., `"0"` instead of `0`). Values that can not be converted are simply discarded.

All the parsed data are used to fill an internal data structure, consisting in a list of **Operation** components, that correspond to the API operations. Each Operation component contains the corresponding endpoint, the HTTP method, and the schemas for input and output data, encapsulated in Data Templates.

Each schema is organized in a **Data Template** component, namely a structure containing parameters names, their types, their domain (e.g., the format for strings and the range for numbers), and any optional constraint. Data Templates are *hierarchical*, so they can model both atomic values (e.g., path parameters), and compound data (e.g., JSON objects). The (user-defined) testing strategy will be responsible to instantiate a Data Template, which means to assign a value to each data element of the template. Peculiar automated testing strategies might also request changes in the Data Template, e.g., to add or remove properties in compound objects, in case the objective of testing is to try with requests that clearly violate the specification.

A test case may execute more interactions (i.e., request and response pairs), that are sequentially related. In this respect, RestTestGen provides a **Test Sequence** component, that is a list of interactions. The latter are modeled with a **Test Interaction** component, containing all the information to test an operation as a reference to the Operation component, including an instance of Data Template with all the concrete data to be used for the test.

Test Sequences are used by the **Test Runner** component, that is responsible for executing the sequence of Test Interactions contained in the Test Sequence on the API. The test output (from HTTP responses) are saved in the corresponding Test Interaction. The Test Runner supports authentication (e.g., API key or bearer token).

Selecting a clever ordering of operations to test is crucial to carry out effective Test Sequences. To help researchers in the definition of the most effective operations ordering to use, RestTestGen provides the **Operation Dependency Graph** (ODG in short) component, that captures producer-consumers dependencies among operations, according to the parsed specification. Nodes represent operations, and directed edges connect two nodes when the output of the source operation can be used as input for the target operation. For instance, consider an operation A with an input parameter named `p`, and an operation B with the same `p` parameter as output. In this case, the ODG will contain a directed edge $A \xrightarrow{p} B$.

The last core component is the **Dictionary**. Dictionaries are available for storing and retrieving values that are observed when testing an API, along with their metadata (e.g., in which operation they have been observed). A *global* Dictionary captures all the values observed during a testing session. Instead, *local* Dictionaries can be deployed for storing values observed in a smaller set of Test Interactions.

III. Extensible Components

Extensible components are a collection of abstract and concrete classes, for which researchers might provide a new concrete implementation to realize a novel testing algorithm, possibly mixing them with RestTestGen core components.

A crucial contribution of a testing approach is the order in which operations are tested [16], [19]. To this aim, RestTestGen provide the **Operation Ordering**, a component with the responsibility to decide the order of the operations in a Test Sequence. There are two variants of Operation Ordering. In a *static* ordering, the Test Interactions are sorted before staring the execution of the Test Sequence. In a *dynamic* ordering, the order in Test Sequence is filled and revised while being tested, because the next operation to test (i.e., the next Test Interaction) depends on the outcome of the previous ones.

An out-of-the-box implementation of Operation Ordering is available, the *Random Ordering*. Researchers can provide further concrete implementations for this component with novel algorithms to deliver their custom ordering.

When instantiating the Data Template, the **Input Value Provider** component is in charge of computing a value for each schema parameter.

The framework already contains concrete single-strategy implementations of Input Value Provider. They are:

- *Enum Value Provider*, returns a random value among those defined as valid enum values in the specification;
- *Example Value Provider*, returns a random value among those in the examples;

- *Default Value Provider*, returns the default value for a parameter;
- *Dictionary Value Provider*, picks a value form the dictionary, if a value for a parameter with the same name was already observed in the testing session and, thus, available in the dictionary;
- *Random Value Provider*, generates a random value according to the parameter schema from the specification.

Furthermore, the framework contains concrete multi-strategies Input Value Providers, that combine multiple existing single-strategies. They are:

- *Random Selector of Input Value Provider*, randomly chooses a single-strategy Input Value Provider among those available and compatible for an input parameter;
- *Enum and Example Priority Input Value Provider*, enum and example values might be more likely to be effective than alternatives. So, this multi-strategies provider chooses providers for enum and example values with a very high probability, and the remaining single-strategies providers with lower probability.

Researchers can deliver their own input generation algorithms by contributing additional concrete implementations of Input Value Provider.

Another fundamental feature that a testing framework should provide is the possibility to dynamically change (i.e., *mutate*) a value for an input parameter. Indeed, RestTestGen has the **Mutation Operator** component, that researchers can implement for defining custom mutation operators.

The **Oracle** component is in charge of making decisions on the correct execution of a Test Sequence (a test case). The outcome of an Oracle evaluation is a **Test Result**, which can be `PENDING`, in case the test case has not been run yet; `PASS`, if the test case has passed; `FAIL`, if the test case has failed; `ERROR`, if the test encountered an error during the execution; and `UNKNOWN`, if the oracle is not able to make a decision.

RestTestGen already includes two implementation of Oracles of general validity. The *Status Code Oracle* classifies a test case as `PASS` if the HTTP response contains a successful status code (`2XX`). Conversely, the test case is a `FAIL` when a server error status code (`5XX`) is observed. In case of a client error status codes (`4XX`), the oracle classifies the execution as `UNKNOWN` because the status code is not informative enough to make a proper decision (the error could be due to a wrong input). The *Schema Validation Oracle* matches the content of an HTTP response with its schema as it is defined in the OpenAPI specification. The test is classified as `PASS` if the response is valid according to the schema, and `FAIL` otherwise.

A further component provided by RestTestGen is the **Writer**, used to write Test Sequences to file. The framework provides the following two implementations of Writer:

- *JSON Report Writer*, emits the report for an executed Test Sequence to file in JSON format, including the HTTP request and response of each interaction, and the outcome of oracles. This file format is meant to be easily parsed and facilitate chaining and integration of this testing framework with other tools;
- *REST-assured Writer*, emits the automatically generated test case as a test case in Java using the REST-assured testing library [20].

Finally, the **Strategy** component is the entry point for the framework, and it represents where the testing strategy business logic should be implemented. A Strategy consists of the integration of the framework components, possibly after having been extended and/or customized.

IV. Concrete Implementation: Nominal and Error Testing Strategies

In this section, we show how two automated test case generation strategies, the Nominal Tester and the Error Tester [16], have been implemented using the framework components.

A. Nominal Tester

Strategy. The goal of the nominal testing strategy is to successfully test all the operations in a REST API according to their definition. To this aim, a custom operation ordering based on the ODG is defined. For all the other responsibilities (input value generation, oracles, and writers) this strategy relies on commodity components available in the framework.

Operation Ordering. To generate nominal test cases, the concrete implementation of the Operation Ordering is a *dynamic* strategy based on the ODG. The next operation to test is selected consulting the ODG, by picking the operation that is still untested, and which minimizes the number of unsatisfied producer-consumer edges. Additionally, the order is adjusted according to CRUD semantics (e.g., delete operations are the last to be tested), to avoid CRUD conflicts (e.g., trying to read a deleted resource).

Input Value Provider. The *Enum and Example Priority Input Value Provider* is used to generate input values, to benefit from all the available single-strategy providers, with higher priority to *Enum* and *Example* values.

Oracles. Test case outcomes are evaluated by the two available oracles. The status code is verified by the *Status Code Oracle*, and the response format is verified by the *Schema Validation Oracle*.

Writers. Java test cases are written by the *REST-assured Writer*. Test Sequences and their outcomes are also written in JSON for logging purposes by the *JSON Report Writer*.

B. Error Tester

The error tester aims at testing a REST API with scenarios that violate its specifications, to try and reveal defects in handling anomalous or wrong requests.

Strategy. The error testing strategy starts from the Test Interactions that the Nominal Tester successfully created (i.e., with `2XX` status code). They are subject to custom mutation operations defined by this strategy. Additionally, a custom ordering and a custom oracle come with this strategy. The remaining components (the writers and one oracle) are those available in the framework.

Operation Ordering. For each nominal Test Sequence of length n, all the sub-sequences are computed starting from the first interaction of the original sequence, having incremental length from 1 to n. Mutation Operators are applied to the last interaction of each of these sequences, when compatible.

Mutation Operators. Three concrete implementations of Mutation Operators are provided:

- *Missing Required*, removes a mandatory input parameter from an Operation to produce a malformed input. Only applicable to parameters marked as `required` in the specification.
- *Wrong Input Type*, changes the type of an input parameter. e.g., an `integer` parameter is assigned a `string` value. The Random Input Value Provider generates a random value of new type.
- *Constraint Violation*, mutates the value of a parameter to make it violate the constraints in the specification. For instance, a string longer than the maximum length, by appending random characters at the end of the original string.

Oracles. The commodity Schema Validation Oracle is integrated to check the validity of the response format. The custom *Error Status Code Oracle* is implemented to verify the test outcome of error scenarios, based on the assumption that a wrong request should to be rejected by a correct REST API. If a request is rejected (status code `4XX`), the test is classified as a `PASS`. Otherwise, if the request is accepted (status code `2XX`) the test is a `FAIL`. In case a server error is observed (status code `5XX`), the test is also a `FAIL`.

Writers. Both the available *JSON Report Writer* and the *REST-assured writer* are integrated.

V. Tool Adoption

In this section, we present execution results for the nominal and error testing strategies. Comparisons with other tools are available in literature [18], [19].

A. Case Study

The Bing Maps REST API is used to perform tasks with the map service from Microsoft, such as creating a static map with pushpins, geocoding an address, or creating a route [21]. In particular, we are going to test five Bing Maps services: *elevations* (4 operations), *imagery* (10 operations), *locations* (5 operations), *route* (15 operations), and *timezone* (4 operations). This same case study was adopted by Wu et al. [22], who manually written the OpenAPI specifications (following the official documentation) because it was not publicly available.

B. Experimental Procedure

First, we configured the authentication for RestTestGen to use the API key that we manually got from the Bing Maps developer platform. Then, we launched the nominal and error strategies on each Bing Maps service.

To control the impact of non-deterministic choices of testing strategies (e.g., the random generation of parameter values), each service have been tested 10 times independently.

TABLE I
Results for nominal and error testing strategies.

		Nominal testing		Error testing		
Service	**Ops**	`PASS`	`FAIL`	`PASS`	`FAIL 2XX`	`FAIL 5XX`
Elevation	4	4.0	0.2	4.0	0.2	4.0
Imagery	10	5.3	4.7	5.3	4.4	0.4
Locations	5	4.4	0.0	4.0	4.4	0.0
Route	15	9.0	1.8	9.0	6.2	0.5
TimeZone	4	4.0	0.0	4.0	3.9	0.4

During the experiment, we collected the results reported by the two status-code-based oracles. For nominal testing, we collected the number of operations for which the strategy could generate at least one valid request (oracle result: `PASS`), and the number of operations for which the strategy could trigger at least one server error (result: `FAIL`). For error testing, we collected the number of operations for which the strategy could generate at least one erroneous request accepted as valid by the server (result: `FAIL` with status code `2XX`), and the number of operations for which the strategy could trigger at least one server error (result: `FAIL` with status code `5XX`).

C. Results and Considerations

Table I reports empirical results for nominal and error testing strategies. Results are in decimal format because they are the average over ten executions.

Despite the OpenAPI specification of this REST API focuses mostly on input data and it specifies the format of output data only partially, the *dictionary* is dynamically updated even with unexpected output data, to steer the automated testing of subsequent operations. Consequently, more than 26 operations are successfully tested over the 38 available operations by the nominal testing strategy. Additionally, this strategy could elaborate faulty executions for 7 operations.

The error testing strategy revealed more than 18 operations that accepted malformed requests as valid. Furthermore, it could trigger server-side errors in 4 (out of 5) services. The error testing strategy shown to be more effective than the nominal strategy in triggering server-side errors. In fact, it could obtain `5XX` status codes for 4 operations in the *elevation* service (the nominal testing strategy could for only one operation and in only 2 executions), and in the *timezone* service, for which the nominal testing strategy could not.

VI. Conclusion

Despite literature contains presentations of several tools to automatically generate test cases for REST APIs, to the best of our knowledge, none of them delivers an extensible framework to guide the implementation of novel, user-defined, testing strategies as extension of abstract components and on top of existing commodity features.

We presented RestTestGen, a framework to support and facilitate researches, practitioners, and developers in implementing novel REST API testing strategies. RestTestGen provides a collection of core (ready-to-use) components and extensible (customizable) components to speed up the development of research prototypes and testing tools. RestTestGen is open-source and available on GitHub [23].

REFERENCES

[1] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000, vol. 7.

[2] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, “Are rest apis for cloud computing well-designed? an exploratory study,” in *Service-Oriented Computing*, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds. Cham: Springer International Publishing, 2016, pp. 157–170.

[3] A. Belkhir, M. Abdellatif, R. Tighilt, N. Moha, Y.-G. Guéhéneuc, and E. Beaudry, “An observational study on the state of rest api uses in android mobile applications,” in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019, pp. 66–75.

[4] simplecloud.info. SCIM. [Online]. Available: http://www.simplecloud.info/

[5] Open Bank Project. OBP Middleware. [Online]. Available: https://www.openbankproject.com/openbankingmiddleware/

[6] A. Arcuri, “RESTful API automated test case generation with Evomaster,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[7] API Fuzzer, “API Fuzzer,” https://github.com/KissPeter/APIFuzzer.

[8] Fuzz-Lightyear, “Fuzz-Lightyear,” https://github.com/Yelp/fuzz-lightyear.

[9] TnT-Fuzzer, “TnT-Fuzzer,” https://github.com/Teebytes/TnT-Fuzzer.

[10] V. Atlidakis, P. Godefroid, and M. Polishchuk, “RESTler: Stateful REST API fuzzing,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[11] S. Karlsson, A. Čaušević, and D. Sundmark, “QuickREST: Property-based test generation of OpenAPI-described RESTful APIs,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 131–141.

[12] N. Laranjeiro, J. Agnelo, and J. Bernardino, “A black box tool for robustness testing of REST services,” *IEEE Access*, vol. 9, pp. 24 738–24 754, 2021. [Online]. Available: https://doi.org/10.1109/ACCESS.2021.3056505

[13] S. Segura, J. Parejo, J. Troya, and A. Ruiz-Cortés, “Metamorphic testing of RESTful web APIs,” *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.

[14] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes, “Specification and automated analysis of inter-parameter dependencies in web APIs,” *IEEE Transactions on Services Computing*, pp. 1–1, 2021.

[15] W. Huayao, X. Lixin, N. Xintao, and N. Changhai, “Test coverage criteria for restful web apis,” in *Proceedings of the 44th International Conference on Software Engineering (ICSE ’22)*, 2022, pp. 1–12, (to appear).

[16] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, “Automated black-box testing of nominal and error scenarios in restful apis,” *Software Testing, Verification and Reliability*, Jan. 2022. [Online]. Available: https://doi.org/10.1002/stvr.1808

[17] swagger.io, “swagger-parser,” https://github.com/swagger-api/swagger-parser.

[18] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, “Empirical comparison of black-box test case generation tools for RESTful APIs,” in *21st IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2021, Luxembourg City, Luxembourg, September 27 - September 28*, 2021.

[19] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for rest apis: No time to rest yet,” 2022.

[20] J. Haleby. REST-assured. [Online]. Available: http://rest-assured.io/

[21] Microsoft, “Bing Maps REST API,” https://docs.microsoft.com/en-us/bingmaps/rest-services/.

[22] H. Wu, L. Xu, X. Niu, and C. Nie, “Combinatorial testing of restful apis,” in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2022.

[23] “RestTestGen,” https://github.com/SeUniVr/RestTestGen.