Automated Blackbox and Whitebox Testing of RESTful APIs with EvoMaster

Andrea Arcuri
Kristiania University College and Oslo Metropolitan University

July 15, 2020

1 Abstract/Intro

RESTful APIs are very popular in industry, especially when developing enterprise systems using a microservice architecture. Testing such APIs is challenging, as tests will be composed of not only HTTP calls, but also settings of the environment, like databases. Different *blackbox* testing techniques have been shown to easily find real faults in many RESTful APIs, with very little human effort from software engineers. However, *whitebox* techniques could lead to much better results, although having an up-front cost for the engineers. In this paper, we report on the use of the open-source tool EvoMaster, on eight RESTful APIs. We show how EvoMaster can be used to automatically generate test cases that can find several bugs, even when using a naive blackbox approach. When enhancing the search with whitebox information, significantly better results are achieved. However, there are several challenges that need to be taken into account when an engineer wants to use a tool such as EvoMaster to test their projects.

2 RESTful APIs

A RESTful API is a web service that exposes functionalities over a network, communicating over HTTP. Many companies use RESTful APIs to expose their services, like Twitter, Google, Amazon, etc. Furthermore, to avoid all the downsides of monolithic applications, modern enterprise applications are built using a *microservice* architecture [1], where web services like RESTful APIs play a major role.

A RESTful API will define a series of *resources* that are identified with URIs. Such resources can then be manipulated with actions based on the semantics of HTTP verbs, like `GET`, `POST`, `DELETE` and `PUT`. During the handling of a HTTP request, the API might need to read/write data from a database, and communicate with other web services.

Testing a RESTful API is challenging. Not only one needs to craft HTTP messages for the different endpoints, but also choose the right query parameters (i.e., the "?" key-value pairs in the URL), URL path-element parameters and body payload messages (e.g., JSON/XML payloads in `POST`/`PUT` requests). The response to a `GET` might depend on the state of the database, which might need to be properly set by previous `POST`/`PUT` requests.

To be able to automatically test a RESTful API, there is the need to know what the URIs of the available resources are, and what the types and names of all parameters and HTTP body payloads are. Sending random bit-strings to a listening TCP socket would likely lead to either invalid HTTP messages, or messages related to resources that do not exist on the server (i.e., resulting in returned 404 HTTP Not-Found status codes). For documentation, it is a common practice to provide a *schema* of what is available in a RESTful API. There are different formats to specify the schema of a RESTful API, where OpenAPI/Swagger seems to be the most popular. For example, *APIs.guru* currently lists more than 500 public APIs with available schemas. Although schemas can be written manually, there are also tools that can automatically generate them based on the source code of the API.

3 Automated Testing

Manual testing of RESTful APIs is a common practice in industry (e.g., with tools such as Postman and REST-Assured). However, given a schema written in OpenAPI/Swagger, test cases can be *automatically* generated by tools crafting and executing valid HTTP calls. This would be very valuable, as writing test cases manually can be expensive and error-prone [2]. An automated testing tool would just need to open a TCP connection toward an up-and-running RESTful API, and then send valid HTTP messages based on the information obtained from the schema (e.g., which are the available endpoints, and what parameters they do require).

When using only information from the schema and no access to the source code, this type of testing can be considered as *blackbox*. Different "coverage criteria" can be defined on the schema of an API [3], like for example trying to cover different HTTP status codes on each endpoint. This type of blackbox testing can be quite effective at finding faults. For example, Atlidakis *et al.* [4] reported on the blackbox testing of the RESTful API of *GitLab*, finding 28 faults in it. Ed-Douibi *et al.* [5] found faults in 37 out 91 public APIs they applied blackbox testing on. Similarly, Segura *et al.* [6] found 11 faults in the APIs of Spotify and YouTube.

However, *"the limitations of blackbox random testing are well-known. For instance, the then branch of the conditional statement "`if(x==10) then`" has only one in* 2^{32} *chances of being exercised if x is a randomly chosen 32-bit input value. This intuitively explains why random testing usually provides low code coverage"* [7]. On the other hand, *whitebox* techniques can overcome these limitations, by analyzing the source code of the API. Different techniques like *Dynamic Symbolic Execution* [8] and *Search-Based Software Testing* [9] have been shown to be highly successful at generating high coverage test cases, in many different testing contexts.

Surprisingly, despite the practical importance of automated test case generation for RESTful APIs, to the best of our knowledge, only our open-source EvoMaster [10, 11] tool does whitebox testing of RESTful APIs. In this paper, we extended it to support blackbox testing as well, and study what results can be achieved when using these two different modes. Furthermore, we show what impacts those two different modes have on the usability of such test generation tools.

For whitebox testing, EvoMaster uses the MIO [12] evolutionary algorithm (which is a search algorithm specialized for test suite generation, where there are many objectives to optimize for). The fitness function is based on several metrics, such as line coverage, branch coverage, HTTP status coverage and detected faults. Search-based software testing uses several white-box heuristics (by instrumenting the SUTs) to help solving the constraints in the predicates of the source code of the SUTs, like for example the so called *branch distance* [9]. Furthermore, several advanced techniques such as *testability transformations* are used in EvoMaster [13] to exploit whitebox information from the SUT (e.g., to check when test inputs are matched against regular expressions or used to instantiate date objects) to achieve higher code coverage.

During the evolutionary search, EvoMaster does direct HTTP calls toward the SUT to evaluate the fitness of the evolved test cases. At the end of the search, the outputs of the tool are test suite files (e.g., in JUnit format).

As an automated oracle (i.e., an automated way to detect if the output of a test is faulty), EvoMaster checks for 500 status code in the HTTP responses, for each API endpoint. As the same endpoint can fail in different ways, EvoMaster keeps track of the last executed statement in the business logic of the API. In this way, EvoMaster can distinguish between two (or more) different faults resulting in a 500 for the same endpoint.

For blackbox testing, EvoMaster relies on random testing, where we aim at maximizing the HTTP status coverage. In other words, we want to have test cases that can exercise every single endpoint in different ways, whether it is a success call (i.e., status code 2xx), a server failure (i.e., status code 5xx) or user error (i.e., status code 4xx). The generated random inputs are still valid, as sampled according to the grammar defined by the OpenAPI/Swagger schema of the SUT. The sampling of a new test case in this random testing is exactly the same as the sampling of a new random individual when using an evolutionary algorithm such as MIO [12].

4 Experiments

4.1 Setup

For the experiments, we selected the same SUTs used in our previous work [11]. However, besides open-source APIs, we also added an industrial SUT from one of our industrial partners. Table 1 provides some statistics on such SUTs, where the industrial API is simply referred with the label *industrial*. We provide the source-code of all these APIs (but the industrial) on *GitHub* at *https://github.com/EMResearch/EMB*.

We ran EvoMaster on all those APIs in two modes: *blackbox* (BB) and *whitebox* (WB). Each experiment was repeated 10 times, to take into account the randomness of the algorithms. Given eight SUTs, this led to a total of $8 \times 2 \times 10 = 160$ runs of EvoMaster. The stopping criterion was 1 million HTTP calls per run.

Depending on the API and the hardware, this took roughly up to 10 hours per run. Using a fixed amount of HTTP calls as stopping criterion would allow

Table 1: RESTful APIs used in the empirical study. We report the number of Java/Kotlin classes, lines of code (LOC), and number of HTTP endpoints.

Name	Classes	LOC	Endpoints
catwatch	69	5442	13
features-service	23	1247	18
industrial	75	5687	20
proxyprint	68	7534	74
rest-ncs	9	602	6
rest-news	10	718	7
rest-scs	13	862	11
scout-api	75	7479	49
Total	342	29634	198

Table 2: Results of the experiments, for both the blackbox (BB) and whitebox (WB) configurations. We report the average (out of 10 runs) test suite size (counted in number of HTTP calls), the average percentage of endpoints for which we got a successful 2xx HTTP status code, server error 5xx code, the average number of distinct found faults, and line coverage measured with IntelliJ.

SUT	Size		#2xx		#5xx		Faults		Coverage	
	BB	WB	BB	WB	BB	WB	BB	WB	BB	WB
catwatch	20.0	335.5	46.2%	53.8%	38.5%	46.2%	5.0	19.9	25%	37%
features-service	33.4	381.5	38.9%	97.0%	72.2%	77.8%	13.0	21.2	48%	73%
industrial	20.0	281.9	5.0%	19.0%	0.0%	95.0%	0.0	26.1	7%	17%
proxyprint	195.9	663.5	56.8%	59.5%	37.8%	41.0%	28.0	38.1	42%	49%
rest-ncs	10.0	239.6	100.0%	100.0%	0.0%	0.0%	0.0	0.0	55%	97%
rest-news	11.0	143.0	28.6%	68.6%	14.3%	28.6%	1.0	2.0	57%	80%
rest-scs	11.0	611.5	100.0%	100.0%	0.0%	0.0%	0.0	0.0	58%	91%
scout-api	134.3	636.8	53.1%	77.9%	67.3%	67.3%	33.0	54.8	30%	33%

the replication of this study regardless of the hardware involved.

4.2 Results

Table 2 shows the results of these experiments, averaged over 10 runs. However, to be able to measure code coverage for blackbox testing, we had to manually modify the generated tests to run the API in the same process of the tests. As far as we know, this should have had no impact on the reported achieved coverage. The coverage was then measured using the tools of the IntelliJ IDE. To make comparisons fair with the whitebox tests (which are self-contained, and start/stop the APIs automatically by themselves), their coverage was measured with IntelliJ as well, instead of just relying on the coverage metrics provided by EvoMaster. As this process of running tests and collect their coverage results was manual, we only applied it on the *best* test suite generated from each of the 10 repetitions. For whitebox testing, the *best* here is identified as the test suite with highest aggregated coverage (e.g., lines, branches and HTTP statuses) out of the 10 generated. For blackbox testing, we only looked at the highest HTTP coverage for this selection, as blackbox testing does not collect any info on code coverage.

On every single API, whitebox testing led to better results. However, the final test suites are smaller for blackbox testing. Note that the final test suites do not contain 1 million HTTP calls, as only the tests that contribute to coverage are retained. Also note that each generated test case might contain a variable number of HTTP calls (e.g., from 1 to 10).

It is not surprising that a technique aiming at maximizing code coverage does achieve better coverage. Improvements can be very large, e.g., from 55% to 97% for *rest-ncs*. Regarding fault finding, the same endpoints can fail in several different ways due to different faults. This can be seen in Table 2 by looking at *scout-api* for example, where the number of endpoints returning a 500 are the same between blackbox and whitebox, but this latter finds more distinct faults. In these experiments, *more than 150 distinct faults were automatically detected*.

The number of endpoints that fail with a 5xx are very similar between whitebox and blackbox testing. This is due to input validation. When provided with an invalid input, a HTTP endpoint should return a code from the 4xx family to represent a user's error. However, most of these endpoints do poor input validation, where invalid inputs are not discarded. Those invalid inputs are then likely going to throw exceptions in the code of the API (e.g., null-pointer exceptions). When a business logic of an endpoint throws an exception, the HTTP server will not crash, but rather craft a HTTP response with status code 500.

One surprising result from Table 2 is that it seems that there are more invalid inputs than valid ones, and so *it is easier to find faults than generating valid inputs*. In these cases, even a naive blackbox technique can easily find this kind of bugs. For many endpoints (75 out of 198, i.e., 37%), it was not possible to create HTTP calls for which a success 2xx status code was returned. All calls either returned a 4xx or 5xx error code. Test cases with 2xx status codes do not directly reveal faults (unless an automated oracle is available to validate the correctness of the returned results, if any). However, they are still very valuable, as they can be used for *regression testing* in a Continuous Integration server (e.g., Jenkins).

Regarding code coverage, there are cases in which EvoMaster is still not particularly effective. For example, compared to a naive blackbox approach, coverage results for *scout-api* only improved by 3%, while for *proxyprint* it improved by only 7%. In both cases, coverage is less than 50%. Although the use of whitebox testing more than doubled the code coverage on the industrial API, such coverage is still lower than 20%. Many whitebox heuristics could be designed to improve performance (e.g., more advanced *testability transformations* [13]). For example, EvoMaster analyses the interactions with SQL databases, and use such information to improve the test generation [14]. However, it is clear that more needs to be done.

One further issue to consider is that the implemented blackbox technique in EvoMaster is rather basic. More advanced blackbox techniques could be designed. Whether such other blackbox techniques could be more effective than search-based whitebox testing is a technical possibility. However, one of the major strengths of search algorithms is their adaptability and handling of multiple objectives [9]. Such other blackbox techniques could be integrated and enhanced in an evolutionary search, as yet another objective to optimize for.

5 Usability

Although the experiments in this paper show that whitebox testing led to better results (i.e., higher code coverage and higher number of detected faults), there are still *practical* benefits for blackbox testing:

- The API could be written in any programming language, e.g., Java, Python, Go and C#. On the other hand, a whitebox approach could only be applied to programming languages that the tool would support.
- The API can be run on a remote machine, without the need of access to source code or executables.
- For an engineer, starting to use a blackbox test generation tool on their API could just take a couple of minutes, as it is just a matter of providing the URL of the API, and the information of where its schema is located.

This last point is critical for *usability*. To be able to use EvoMaster in whitebox mode, engineers need to write a configuration *driver* [10, 11]. These are small classes used to specify how to *start*, *stop* and *reset* the API (e.g., delete all data in the used databases, for which EvoMaster provides easy to use support functions). The whitebox code analyses and instrumentations are then done automatically when the API starts. Although the amount of code needed to be written is small, it is still a non-zero manual cost. *A practitioner might not want to try out a tool coming from academia if it requires an up-front cost.* At any rate, the ability to start, stop and reset the SUT is essential if the generated tests are then to be used for regression testing. Tests should not fail based on the order of their execution, or based on the state of the database (if any), or simply fail because the SUT is currently not up and running.

Blackbox testing enables the generation of test cases with near-to-zero manual cost (i.e., just download a testing tool and give the API schema as input). This blackbox testing will give worse results (i.e., lower coverage and fewer faults detected compared to a whitebox approach, and the generated tests would not be viable for regression testing), but they would come for "free". And such initial results, if positive, might be enough to convince the engineers to write the drivers. Furthermore, if an engineer is working on an API with some already existing end-to-end tests, it is likely that all the code to start/stop/reset the API is already available.

When testing a web/enterprise application via its GUI, a blackbox approach can be very useful, because any user can do it, as long as it is just a matter of clicking buttons and filling text forms. These testers do not necessarily have to be engineers with strong coding skills, or any coding skill at all (of course, you still want to have QA professionals, if possible). Having someone else to test a system besides its developers is a common and useful practice. On the other end, testing a RESTful API requires knowledge of HTTP, and at least some minimal coding skills, as you need some sort of scripting to make the HTTP calls and create the proper JSON/XML body payloads for the `POST`/`PUT` requests. It is reasonable to assume that, in many cases, it ends up to the developers themselves to do the testing of their APIs. In such cases, there is no particular limitation that prevents the use of whitebox techniques when testing single APIs in isolation.

6 Test Automation Challenges

Tools such as EvoMaster can already be used in practice to find bugs in existing RESTful APIs. But there are still open challenges that need to be addressed to make such types of tools common among practitioners [2]. The main ones are the handling of the *environment* of such APIs, like *databases* and *external services*, and how to design effective *automated oracles*.

6.1 Databases

If a RESTful API is connected to a database, actions like `POST` and `PUT` could modify the state of such database. Furthermore, to be able to properly function, such databases must be up and running when the SUT is tested.

For a blackbox testing of a remote web service, this is a major challenge. If one does not have control over the database, there would be no guarantee on its content, which could be different and changed at any time. Furthermore, tests could become *flaky*, i.e., running the same test again could lead to different results, if those results depend somehow on the state of the database.

When developers do whitebox testing of their APIs, they can have control on the databases. For the JVM, there are embedded SQL databases such as H2 and Derby. If one needs something like Postgres or MySQL, those can be downloaded/started programmatically via Docker.

Still, even in the case of whitebox testing, there are challenges to address. To make tests independent, in EvoMaster we clear the state of the database (just the data, and not the schema of the tables) at each test execution. We provide several utility functions (as a published library) for different databases that the developers can call when implementing the *reset* method in the EvoMaster's driver. Still, starting from an empty database could pose several challenges when we need to test a `GET` endpoint that depends on a complex data setup (which might require a long sequence of prior `POST` requests to create the needed data).

To address these challenges, we enhanced EvoMaster to be able to analyze the state of the database, intercept and analyze all SQL commands the SUT does, and also added the generation of test data directly into the database as part of the generated test cases [14]. This novel approach has shown promising results [14], but more needs to be done to handle the different edge cases in SQL databases, and provide better, more efficient heuristics for test generation.

6.2 External Services

A RESTful API could depend on communications with other RESTful APIs. This is a typical case in microservice architectures [1]. Unfortunately, this severely complicates the testing of such APIs.

Similarly to the issue with databases, in blackbox testing of a remote API there would be no control on the used external services. Therefore, tests have high chance to be *flaky*. Interactions with such external services would not only depend on their implementation (which could change), but also on their state (e.g., they could use their own databases).

This is a major challenge for whitebox testing as well. Even if a developer had full control on all those external services (e.g., they are all part of the same microservice system), setting up and start several different web services (which can use their own databases) used by the SUT just to test this latter could be cumbersome. A common approach [2] in industry is to *mock* those services, using for example libraries such as WireMock. This enables to start a HTTP server directly from the tests, where the tests can specify what should be returned at each different type of HTTP incoming request from the SUT (e.g., based on regex matching of the URLs of the requests).

Writing this kind of mocks manually can be very time consuming [2]. Generating the mocks automatically is a major research challenge that needs to be addressed to be able to test this kind of APIs. Currently, EvoMaster has no support for it, but there is ongoing work to address it. All the SUTs in the empirical study were not directly relying on external web services.

6.3 Automated Oracles

To find bugs, there is the need for an *automated oracle* that can verify whether a given output is correct or not. Currently, EvoMaster checks for 500 error status messages. As an endpoint could fail in different ways due to different bugs in different parts of the code, one advantage of whitebox over blackbox testing here is that we can detect which was the last executed statement in the SUT before a failure. This can help distinguish between different bugs in the same endpoints.

As a further oracle, EvoMaster also checks if the responses of the SUTs are conforming to the OpenAPI/Swagger schema of the SUT (this was not discussed in the empirical study, as all those schemas were automatically generated, and so any mismatch there would had pointed only to bugs in the schema generator tools, and not the SUT itself). This applies to both whitebox and blackbox testing.

However, not all bugs lead to a crash in the SUT, or a mismatch with the schema. Bugs could lead to inconsistent data in the database for example. Or filtering operations on collections that wrongly return more data than no applied filter (this is for example the type of automated oracle investigated in [6]). To make this kind of test generation tools more appealing to developers, there is the need for more research on designing novel automated oracles to expand the types of faults that can be automatically detected.

Another relevant issue is that not all faults have the same severity. A wrongly returned 500 status code due to a faulty input validation can be a problem for a publicly available service on the internet. This is especially the case when paying customers might think it is something wrong with the service (and so waste time in reporting the problem), instead of fixing how they used it (e.g., 4xx responses can contain info on what the users did wrong). However, it might be less of a problem for a private API in a small system, where the “users” might be just a small team of developers that communicate with each other on a daily base. Furthermore, crashing (e.g., 500 code) on a “valid” input, that should have been successfully handled (i.e., a 2xx status), is a much more serious bug than wrongly returning a 500 (server error) instead of a 4xx (user error). But how to automatically determine if an input was supposed to be “valid”? Given a set of faults found by a test generator tool such as EvoMaster, there is the need for a way to automatically classify and rank the severity of those faults. This is a problem for practitioners, especially if those tools find tens/hundreds of bugs, and do not have time/resources to investigate/fix all of those bugs.

7 Conclusion

In this paper, we have carried out experiments to compare blackbox and whitebox testing of RESTful APIs, using automated test case generation techniques. When it comes to code coverage and fault detection, in our experiments whitebox-based test generation led to significantly better results. However, whitebox testing comes with a *usability* cost, as setting it up requires more effort from the developers than just running a blackbox test generation tool. Therefor, blackbox testing can still be useful for practitioners that want to try out this kind of test automation tools, but that do not want to invest resources before seeing some practical results.

In our empirical study, it was possible to automatically find 150 faults. This provides further evidence that automated test case generation techniques might already be of use for practitioners in industry. However, there are still many research challenges that need to be addressed to achieve a better automation of testing RESTful APIs, like the automated handling of external services and the classification of the found faults. This will be needed to spread the use of this kind of automated test generation techniques among practitioners. Automated test generation tools have the potential to significantly help practitioners in assuring the quality and correctness of their implemented systems.

To the best of our knowledge, our EvoMaster tool is currently the only whitebox test generation tool for RESTful APIs. EvoMaster is released as open-source on *GitHub*. Website at: *www.evomaster.org*

Acknowledgment

This work is funded by the Research Council of Norway (project on Evolutionary Enterprise Testing, grant agreement No 274385).

References

[1] S. Newman, *Building Microservices.* " O'Reilly Media, Inc.", 2015.

[2] A. Arcuri, ''An experience report on applying software testing academic results in industry: we need usable automated test generation,'' *Empirical Software Engineering (EMSE)*, pp. 1--23, 2018.

[3] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, ''Test coverage criteria for restful web apis,'' in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* ACM, 2019, pp. 15--21.

[4] V. Atlidakis, P. Godefroid, and M. Polishchuk, ''Restler: Stateful rest api fuzzing,'' in *ACM/IEEE International Conference on Software Engineering (ICSE).* IEEE Press, 2019, pp. 748--758.

[5] H. Ed-Douibi, J. L. C. Izquierdo, and J. Cabot, ''Automatic generation of test cases for rest apis: A specification-based approach,'' in *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC).* IEEE, 2018, pp. 181--190.

[6] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, ''Metamorphic testing of RESTful web APIs,'' *IEEE Transactions on Software Engineering (TSE)*, 2017.

[7] P. Godefroid, ''Random testing for security: blackbox vs. whitebox fuzzing,'' in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007).* ACM, 2007, pp. 1--1.

[8] C. Cadar and K. Sen, ''Symbolic execution for software testing: three decades later.'' *Commun. ACM*, vol. 56, no. 2, pp. 82--90, 2013.

[9] M. Harman, Y. Jia, and Y. Zhang, ''Achievements, open problems and challenges for search based software testing,'' in *IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2015, pp. 1--12.

[10] A. Arcuri, ''EvoMaster: Evolutionary Multi-context Automated System Test Generation,'' in *IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2018, pp. 394--397.

[11] ------, ''RESTful API Automated Test Case Generation with EvoMaster,'' *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, p. 3, 2019.

[12] ------, ''Test suite generation with the Many Independent Objective (MIO) algorithm,'' *Information and Software Technology*, vol. 104, pp. 195--206, 2018.

[13] A. Arcuri and J. P. Galeotti, ''Testability Transformations For Existing APIs,'' in *IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 2020.

[14] ------, ''SQL data generation to enhance Search-Based System Testing,'' in *Genetic and Evolutionary Computation Conference (GECCO).* ACM, 2019, pp. 1390--1398.