OAS DB: A Repository of Specifications to Support OpenAPI Research

Alex Braha Stoll
University of São Paulo - USP
São Paulo - Brazil
alex.stoll@usp.br

Marcos Lordello Chaim
University of São Paulo - USP
São Paulo - Brazil
chaim@usp.br

Tosin Daniel Oyetoyan
HVL
Bergen - Norway
tosin.daniel.oyetoyan@hvl.no

Daniela Soares Cruzes
SINTEF
Trondheim - Norway
daniela.s.cruzes@sintef.no

Abstract

There are many specifications used to describe a Web API. One of the most popular ones is OpenAPI. This specification allows one to describe all the resources that can be accessed and manipulated through a REST Web API. An OpenAPI specification can be used to perform different kinds of analysis and verification of the service implementing the described API. A common challenge faced by researchers, however, is the lack of a standard repository of samples. A repository with annotated samples (e.g., anti-patterns found in each specification) would be a valuable resource because it would help researchers and practitioners to verify new tools and techniques aiming at OpenAPI specifications. Such a repository would also allow direct comparison between different studies that decide to employ it. This paper describes the creation of a repository of annotated synthetic (but realistic) OpenAPI samples.

1 Introduction

A popular choice when building Web systems and APIs is to use the REST (Representational State Transfer) architectural style. Introduced in 2000 [7], it aims to improve the scalability, generality and independence of the components of a software system.

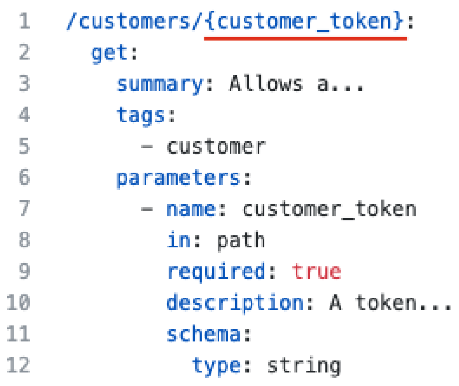
The automatic verification of REST Web APIs is still not a common practice due to the lack of a widely accepted set of best practices and also the absence of tools developed from the ground up to be used with that particular architectural style (many of the available tools are adaptations of solutions created to handle older architectures) [1]. Therefore, many checks that may contribute to the quality and security of these APIs are being done manually and in an inconsistent fashion (or are not even being done due to the high cost of manual analysis).

Another challenge faced by researchers exploring this area is the lack of a standard repository with OpenAPI specification samples. The absence of such a database forces researchers to diverge time from the main objectives of their work into building datasets. Besides that, it also makes the comparison between different studies harder, since the datasets used are generally different.

Considering all the aforementioned facts, it appears to be important the creation of a repository of OpenAPI samples to support researchers doing work in this field. This repository of OpenAPI samples with common defects found in real-world specifications may facilitate studies analyzing and verifying OpenAPI specs, which in turn may result in an increase in the efficiency and efficacy of the studies. Besides that, by supporting researchers who are developing tools and techniques that are helpful in managing software evolution, we believe we will also be making an indirect but meaningful positive impact in this area as well.

This paper describes how such a repository of synthetic and annotated OpenAPI samples could actually look like.

To make it all clearer, let us consider a short example of one such OpenAPI sample with an anti-pattern. In REST APIs, a good practice is to disallow sensitive information in the query string portion of a given endpoint [8]. The reason for that is the lack of encryption of the URI (which the query string is a part of), even when a secure protocol like HTTPS is used. Therefore, the *inclusion* of sensitive data in the query string is an anti-pattern. Figure 1 illustrates an instance of this anti-pattern. It shows an endpoint (*/customers/{customer_token}*) that includes a query string parameter (*customer_token*) that holds sensitive data (a token used to identify a user).



```
/customers/{customer_token}:
  get:
    summary: Allows a...
    tags:
      - customer
    parameters:
      - name: customer_token
        in: path
        required: true
        description: A token...
        schema:
          type: string
```

Figure 1: OpenAPI Segment Including an Anti-pattern

The rest of this paper is structured as follows: section 2 offers the reader straightforward explanations of the main concepts and technologies relevant to this paper; section 3 briefly discusses relevant related works; section 4 details the rationale and steps taken while planning the proposal for OAS DB and building a proof of concept of it; section 5 presents the outcomes on the initial efforts put into building OAS DB; section 6 discusses potential threats to the validity of this work; and, finishing up, section 7 summarizes the results of this work in progress and also presents some ideas for future developments.

2 Background

2.1 API and OpenAPI

Application Programming Interface (API) is a specified set of operations for programmatically interacting with components of a software system. In particular, a Web API is an interface that allows a web system to receive requests from other systems. REST (Representational State Transfer) is one of the multiple architectural styles that can be adopted when implementing a Web API. Introduced in Roy Fielding's doctoral thesis [7], this architectural style defines a set of recommendations with the objective of improving the scalability of interaction between components, the generality of interfaces and the independent deployment of components. REST also aims to allow intermediary components to reduce the latency of interactions, to enforce security constraints and to also be able to encapsulate legacy systems.

OpenAPI[1] is a specification that allows one to describe all the resources that can be accessed and manipulated through a REST Web API. An OpenAPI specification offers a level of detail and formalism that makes it possible not only to automatically deduce properties about the Web API and the software system exposing it, but also to automatically interact with the API.

2.2 REST Patterns and Anti-patterns

A REST pattern is a good practice that should be followed when developing a REST API. On the other hand, an anti-pattern is a bad practice that should be avoided [3]. In the context of REST APIs, good practices are generally the result of following the recommendations of the REST architectural style. As a consequence, having anti-patterns in an API can be detrimental to one or more software quality attributes (e.g., an anti-pattern may cause a service to be harder to maintain over time). One can find in the literature collections of REST patterns and anti-patterns created from surveys of academic works and industry practices (e.g., see Brabra et al. [3]).

3 Related Work

3.1 OpenAPI Tools and Techniques

By reviewing the literature on REST APIs, one can find research tackling issues around designing, testing and analyzing APIs from different angles. In Petrillo et al. [10], a catalog of 73 REST API best practices is proposed; this same catalog is then used to assess the maturity of popular and established REST APIs of major cloud infrastructure providers. In Ed-Douibi et al. [6], the effort is on automatically generating test cases for REST APIs having as the sole requirement a valid OpenAPI specification. The authors introduce a tool capable of detecting different kinds of errors, such as discrepancies between the data an API operation is supposed to return and the data that is actually received from the system under test. As one last example of work being conducted in the field, in the work of

[1] https://www.openapis.org/

Iversen [8] the main objective is to detect vulnerabilities by analyzing - with static and dynamic techniques - specifications describing a REST API.

This survey of the literature points to some interesting facts. Most relevant to this work is the usage of different datasets in many related research projects. One can argue that this is detrimental to advancements in the field for two main reasons: first, researchers are having to spent valuable time building their own datasets; and, besides that, the usage of different data makes the comparison between studies more challenging.

Efforts to build shared experimentation infrastructure seem to have yielded positive effects in other research fields. In Do et al. [5], the many benefits of having shared infrastructure for experimentation, such as cost saving and accelerated improvement of datasets (because different researchers are able to provide feedback and collaborate), are demonstrated. Another project that is worth mentioning is the Defects4J repository. Containing hundreds of bugs from real-world Java programs, it has been successfully shared and improved by different research teams [9].

3.2 Existing API Collections

There do exist directories of APIs (e.g., RapidAPI[2]) and even collections of OpenAPI specifications (e.g., APIs Guru[3]). However, these existing solutions are not a good fit for researchers for two important reasons.

The first one is the lack of annotations in the OpenAPI samples, making it challenging for a researcher to check the performance of a tool tested against these existing solutions. Without annotated samples, it becomes labor intensive to produce metrics because one has to manually analyze every specification touched by the tool under assessment (e.g., to confirm true positives). The second reason is the fact that the focus of these repositories is simply on creating OpenAPI specifications for existing web services, without a concentrated effort (such as in the case of OAS DB) in adding new samples that actually increase the diversity of scenarios covered (both in terms of anti-patterns contained in the repository and in terms of domains covered by its samples).

4 Building a Proof of Concept of OAS DB

The literature review described in section 3 allowed us to become familiar with three essential facets of the different studies: the techniques being used, the datasets being employed and the most common issues found in the APIs considered by each study. Having all this information available, we set out to build a repository of OpenAPI samples that would be immediately compatible with the reviewed studies or at least would be employable after some adaptations to them.

We built the repository by creating a set of OpenAPI specifications describing different kinds of synthetic but realistic APIs. For example, one of our samples describes the API for an e-commerce. Each specification is accompanied by files containing metadata. These files allow one to quickly and automatically check which anti-patterns or issues are present in each specification.

We also intend to develop a tool capable of generating mock servers from the OpenAPI specification samples that are part of the repository. These API servers can be run in the researcher's local machine or be automatically deployed to a compatible cloud provider.

Figure 2 shows an overview of OAS DB. As explained, each OpenAPI specification has a corresponding annotation file. The specifications are also used to generate the mock servers and it is possible to have these API servers optionally deployed to a compatible cloud provider.

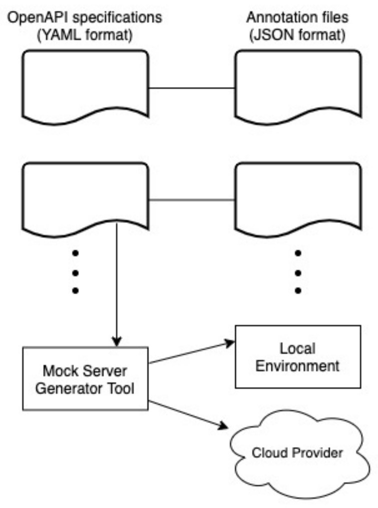

OpenAPI specifications (YAML format)
Annotation files (JSON format)
Mock Server Generator Tool
Local Environment
Cloud Provider

Figure 2: Overview of OAS DB

4.1 Categories of Anti-patterns

To help us in having a diverse set of anti-patterns present in OAS DB, we first devised a list of categories to classify the different kinds of REST API bad practices. To create this set, we had as a foundation both the classical attributes of software quality and also the dimensions and groupings found during our review of

[2]https://rapidapi.com
[3]http://apis.guru

works exploring REST anti-patterns.

We classified each selected anti-pattern as belonging to one or more of the following categories: compatibility, discoverability, understandability and security.

4.2 Anti-patterns Selection

As a proof of concept of OAS DB, we decided to choose one anti-pattern for each category among the bad practices deemed most relevant and / or common in the reviewed literature (see Brabra et al. [3], Ed-Douibi et al. [6] and Petrillo et al. [10]). We also added anti-patterns known to the authors (for example learned through their experiences in the industry) when the bad practice could also be supported by academic or technical literature.

Figure 3 shows an example of a segment of OpenAPI specification that contains an anti-pattern (*Deep path*). This anti-pattern happens when an endpoint includes unnecessary IDs. By unnecessarily requiring the ID of a parent resource to manipulate a given object, the developer makes the API more complex and potentially harder to use. Besides that, it increases the surface an attacker generating fake IDs has to try and find authorization bugs in the underlying service [2].

```yaml
/orders/{order_id}/items/{item_id}:
  get:
    summary: Shows an item of a given order.
    operationId: show_item
    tags:
      - items
```

Figure 3: Segment Including the Deep Path Anti-pattern

4.3 Annotation Files

Each OpenAPI specification present in OAS DB is accompanied by a JSON (JavaScript Object Notation) file. Each annotation file describes all anti-patterns found in the associated specification, including the line in the OpenAPI file responsible for each violation. These annotations provide a way for researchers to automatically verify the efficacy of new techniques and tools.

Figure 4 shows an example of an OAS DB annotation file. The JSON key *violations* has a collection of all the violations present in the corresponding OpenAPI specification. For each violation, the following data is available:

- **type:** this key holds the name of the anti-pattern being described;
- **categories:** to which categories (e.g., *security*) the anti-pattern belongs to;
- **offender:** the segment in the corresponding OpenAPI specification to be blamed for the violation;
- **location:** the line in the corresponding specification file where the violation is found.

```json
{
  "version": "oas-db-0.1",
  "annotationTarget": "ecommerce.yml",
  "violations": [
    {
      "type": "deep_path",
      "categories": ["understandability", "security"],
      "offender": "paths./orders/{order_id}/items/{item_id}",
      "location": 43
    }
  ]
}
```

Figure 4: Example of an OAS DB Annotation File

4.4 Automatic Generation of Mock Servers

OpenAPI specifications are rich enough to allow one to generate a mock server capable of returning static responses (i.e., the same fixed response independent of the request parameters) for real HTTP requests. OAS DB will leverage that capacity and include a tool to allow researchers to generate mock servers (and have them running on their local machine) for the sample APIs included in the repository. This tool will also allow a researcher to deploy the mock server to any cloud provider, as long as an OAS DB adapter for the desired provider is available. We intend to include out-of-the-box an adapter for the Amazon Web Services[4] platform.

To generate the mock servers, we will leverage the popular and mature Prism[5] open-source project. Prism is implemented in the TypeScript programming language and allows one to generate mock servers from an OpenAPI specification versions 2 or 3. For the feature that will allow a researcher to deploy these generated mock servers, we will make use of Docker[6] containers.

We believe that mock server generation will prove to be a very useful feature of OAS DB. As shown in different studies in the relevant literature (e.g., in Iversen [8] and in Atlidakis et al. [1]), some anti-patterns can only be detected when dynamic techniques are employed. In other words, this means that in many cases

[4]https://aws.amazon.com/
[5]https://github.com/stoplightio/prism
[6]https://www.docker.com/

statically analyzing an OpenAPI specification is not enough to detect an issue and having a running server capable of receiving and responding to HTTP requests is essential.

5 Preliminary Results

OAS DB[7] is a proof of concept of what we are envisioning. As of the submission of this article, it has two OpenAPI specifications (and one annotation file for each specification), featuring five unique anti-patterns (the ones listed in Table 1) and a few instances of these types of bad practices across all specifications. Having the purpose of being a proof of concept, it is not the purpose of the current version of the repository to have a large number of samples, but to provide us with a sandbox for experimentation on the ideas here discussed. We intend the finished version of OAS DB to have dozens of samples, featuring many different unique anti-patterns.

Table 1 lists all the anti-patterns present in the current version of OAS DB, along with references in the literature supporting their relevance.

Table 1: Anti-patterns in the Proof of Concept of OAS DB

Anti-pattern name	Supported by
Sequential integers as resource ID	[4]
Deep path	[2]
Sensitive information in the path or in the query string	[8]
Inappropriate HTTP method	[3]
Lack of hypermedia support	[3]

6 Threats to Validity

Given the characteristics of this work, the main validity risk we face is its external validity. To mitigate this danger, we selected anti-pattern categories and types that are representative of both what is generally found in datasets used in research and what one encounters when examining real-world APIs (see Section 4). We will continue to adopt this same strategy as OAS DB grows.

7 Conclusion

In order to improve OAS DB, we plan to work on two fronts. The first is to support a larger number of anti-patterns and to have a greater amount of OpenAPI samples. The selection of the next anti-patterns to be supported will follow the same procedure described at Subsection 4.2. The second front is to start implementing the tooling to generate mock API servers for the samples contained in the repository (as described at Subsection 4.4).

[7]https://github.com/alexbrahastoll/oas-db

References

[1] V. Atlidakis, P. Godefroid, and M. Polishchuk. "RESTler: Stateful REST API Fuzzing". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 748–758. DOI: `10.1109/ICSE.2019.00083`.

[2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. *Checking Security Properties of Cloud Services REST APIs*. Tech. rep. Feb. 2019. URL: `https://www.microsoft.com/en-us/research/publication/checking-security-properties-of-cloud-services-rest-apis/`.

[3] H. Brabra et al. "On semantic detection of cloud API (anti)patterns". In: *Information and Software Technology* 107 (2019), pp. 65–82.

[4] *CVE-2015-8542*. Available from MITRE, CVE-ID CVE-2015-8542. Dec. 2015. URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8542`.

[5] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact". In: *Empirical Software Engineering* 10.4 (2005), pp. 405–435. DOI: `10.1007/s10664-005-3861-2`.

[6] H. Ed-Douibi, J. L. Canovas Izquierdo, and J. Cabot. "Automatic generation of test cases for REST APIs: A specification-based approach". In: *Proceedings - 2018 IEEE 22nd International Enterprise Distributed Object Computing Conference, EDOC 2018*. Stockholm, Sweden, Oct. 2018, pp. 181–190. DOI: `10.1109/EDOC.2018.00031`.

[7] Roy Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California, Irvine, 2000.

[8] P. Iversen. "Specification-based security analysis of REST APIs". MA thesis. Norwegian University of Science and Technology, 2018.

[9] R. Just, D. Jalali, and M.D. Ernst. "Defects4J: A database of existing faults to enable controlled testing studies for Java programs". In: 2014, pp. 437–440.

[10] F. Petrillo et al. “Are REST APIs for cloud computing well-designed? An exploratory study”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9936 LNCS (2016), pp. 157–170. DOI: 10.1007/978-3-319-46295-0_10.