

UNIVERSITY OF SAO PAULO  
SCHOOL OF ARTS, SCIENCES AND HUMANITIES  
GRADUATE PROGRAM IN INFORMATION SYSTEMS

ALEX BRAHA STOLL

**OAS DB: A shared infrastructure to support OpenAPI research**

Sao Paulo

2022

ALEX BRAHA STOLL

**OAS DB: A shared infrastructure to support OpenAPI research**

Research area: Computer Science Methodologies and Techniques

Corrected version contemplating the changes requested by the examiners at 28th March 2022. The original version can be found at a special collection at EACH-USP Library and in USP Digital Library of Theses and Dissertations, in compliance with Resolution CoPGr 6018, 13th October 2011.

Supervisor: Prof. Dr. Marcos Lordello Chaim

Sao Paulo

2022

Authorize the reproduction and dissemination of total or partial copies of this document, by conventional or electronic media for study or research purpose, since it is referenced.

#### CATALOGUING IN PUBLICATION

(University of São Paulo. School of Arts, Sciences and Humanities. Library)

CRB 8 -4936

Stoll, Alex Braha

OAS DB: a shared infrastructure to support OpenAPI research /  
Alex Braha Stoll ; supervisor, Marcos Lordello Chaim. – 2022  
98 p.

Dissertation (Master of Science) – Graduate Program in  
Information Systems, School of Arts, Sciences and Humanities,  
University of São Paulo.  
Corrected version.

1. REST API. 2. Open API. 3. Anti-pattern. 4. Software  
fault injection. 5. Static analysis. 6. Code generation. 7.  
Repository. I. Chaim. Marcos Lordello, supervisor II. Title

**EACH**Escola de Artes, Ciências e Humanidades  
Universidade de São Paulo

## ATA DE DEFESA DE DISSERTAÇÃO

A Comissão Julgadora de Dissertação de Mestrado reuniu-se remotamente no dia 28 de março de 2022, às 14h, para arguir, o(a) aluno(a) Alex Braha Stoll, do Programa de Pós-Graduação em Sistemas de Informação, referente à dissertação "OAS DB: a shared infrastructure to support OpenAPI research". A sessão iniciou-se com a apresentação da dissertação pelo(a) aluno(a), seguida de arguição pelos julgadores, que decidiram pela **aprovação** da Dissertação de Mestrado, conforme avaliação a seguir:

Prof./a. Dr./a. Marcos Lordello Chaim  
Presidente

Resultado: Aprovado

  
(assinatura)

Prof./a. Dr./a. Júlio Cezar Estrella

Resultado: Aprovado

  
(assinatura)

Prof./a. Dr./a. Marco A. Graciotto Silva

Resultado: Aprovado

  
(assinatura)

São Paulo, 28 de março de 2022.

Ferramenta utilizada para participação remota:

Comentários:

A sessão de defesa ocorreu com todos os participantes a distância, conectados por meio do aplicativo Google Meet, com transmissão ao vivo. A sessão virtual foi gravada e está disponível para averiguação. A transmissão ao vivo e a gravação foram interrompidas durante a sessão secreta de julgamento e retomadas para proclamação do resultado.

## **Acknowledgements**

I would like to thank my supervisor, Prof. Dr. Marcos Lordello Chaim, for all the work he put forth helping me shape the research into what it has ultimately become.

I would also want to express my gratitude to Prof. Dr. Daniela Soares Cruzes and to Prof. Dr. Tosin Daniel Oyetoyan for participating in discussions about the research and giving ideas and feedback that proved very valuable.

## Abstract

STOLL, Alex Braha. **OAS DB: a shared infrastructure to support OpenAPI research**. 2022. 98 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2022.

It is common knowledge the great success achieved by the Web in the last decades. Together with the rise of Web systems in general, it came the increase of the number of Web APIs. There are many specifications used to describe an 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 shared validation infrastructure or a standard benchmark. The main contribution of our research is a software artifact — called OAS DB (OpenAPI Specifications Database) — that aims to provide researchers and industry practitioners with a complete solution to streamline the validation of new OpenAPI related techniques and tools. OAS DB is able to generate complete OpenAPI specifications and their corresponding mock implementations. It is also both capable of injecting faults and anti-patterns in these generated specification-s/mock implementations and of indicating — through machine-readable files — which issues and anti-patterns are present in the generated assets. We use OAS DB to assess tools relying on both static and dynamic techniques to detect faults and anti-patterns in OpenAPI specifications. Our results indicate that these tools fail to detect relevant faults and anti-patterns in the synthetic APIs generated by OAS DB, indicating that there is room to improve these tools and the ways in which they are applying static and dynamic analysis techniques. The present work also has as contributions: a) a proof of concept REST API anti-pattern detector (which we call Oasis) and b) the description of a novel REST anti-pattern not described in the literature so far.

Keywords: REST API. OpenAPI. Anti-pattern. Software fault injection. Static analysis. Code generation. Repository.

## Resumo

STOLL, Alex Braha. **OAS DB: uma infraestrutura compartilhada para apoiar a pesquisa envolvendo OpenAPI**. 2022. 98 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2022.

Já é senso comum o grande sucesso alcançado pela Web nas últimas décadas. Junto à ascensão de sistemas Web em geral, veio também o aumento do número de APIs Web. Há muitas especificações usadas para descrever uma API Web. Uma das mais populares é a OpenAPI. Essa especificação permite descrever todos os recursos que podem ser acessados e manipulados por meio de uma API Web REST. Uma especificação OpenAPI pode ser usada para diferentes tipos de análises e verificações do serviço que implementa a API descrita. Um desafio comum enfrentado por pesquisadores, no entanto, é a inexistência de infra-estrutura compartilhada de validação ou de um *benchmark* padrão. A principal contribuição de nossa pesquisa é um artefato de software — chamado OAS DB (*OpenAPI Specifications Database*) — que tem por objetivo fornecer aos pesquisadores e profissionais da indústria uma solução completa para tornar mais eficiente a validação de novas técnicas e ferramentas relacionadas com OpenAPI. OAS DB consegue gerar especificações OpenAPI completas e as suas correspondentes implementações *mock*. É também capaz de injetar defeitos e *anti-patterns* nessas especificações/implementações *mock* geradas e também de indicar — por meio de arquivos processáveis por software — quais defeitos e *anti-patterns* estão presentes nesses arquivos gerados. Ferramentas que usam técnicas estáticas e dinâmicas para identificar defeitos e *anti-patterns* em especificações OpenAPI foram avaliadas usando o OAS DB. Os resultados indicam que essas ferramentas não detectam alguns defeitos e *anti-patterns* relevantes em APIs sintéticas geradas pela OAS DB. Esses resultados indicam que essas ferramentas e o modo como aplicam técnicas de análise dinâmica e estática podem ser melhorados. Este trabalho também tem como contribuições a) uma prova de conceito de detector de *anti-patterns* REST (chamado Oasis) e b) a descrição de um novo *anti-pattern* REST ainda não documentado na literatura relevante.

Palavras-chaves: REST API. OpenAPI. *Anti-pattern*. Injeção de defeitos em software. Análise estática. Geração de programas. Repositório.

## List of Figures

Figure 1 – Interaction with an example REST Blog Application: the user requests (by using the Web Browser) a specific article; the server responds the GET HTTP request with the contents of the article in the HTML format (which will then be rendered into a user interface by the Web Browser)	20
Figure 2 – An example of SOAP message . . . . .	21
Figure 3 – By using the mobile application to create a new article, a POST request with the contents of the article represented as JSON is issued to the server. Note that the server not only responds with a status code indicating success, but also includes the URI of the newly created Web resource (the new article) in the Location header . . . . .	24
Figure 4 – A segment of an OpenAPI specification documenting a particular operation available in the API being described . . . . .	26
Figure 5 – IBM OpenAPI Validator reporting an offence after linting an OpenAPI specification . . . . .	26
Figure 6 – A simple API implemented by using the Sinatra framework, available for the Ruby programming language. . . . .	27
Figure 7 – Multiple containerized applications, each with its own isolated environment but still sharing the operating system kernel via Docker . . . . .	28
Figure 8 – A client requesting an order’s data by providing its integer ID . . . . .	46
Figure 9 – Sensitive information included in the path or in the query string, a cataloged REST anti-pattern . . . . .	48
Figure 10 – Overview of OAS DB . . . . .	50
Figure 11 – An OpenAPI specification seed that describes the kernel of an incident reporting API. . . . .	52
Figure 12 – Sensitive information (the customer token) included in the query string, one of the new anti-patterns we propose. . . . .	53
Figure 13 – A segment of a mock API implementation. This excerpt shows a GET endpoint that is able to retrieve a record when given its ID. . . . .	55
Figure 14 – A segment of a mock API implementation. This excerpt shows a method whose implementation includes injected issues. . . . .	57



Figure 15 – The bulk of the code implementing the embedded key-value store. The implementation leverages Ruby’s Hash data structure. . . . .	59
Figure 16 – Part of the code implementing the data type validation mechanism. . .	60
Figure 17 – An annotation file documenting that the associated OpenAPI specification has one anti-pattern . . . . .	62
Figure 18 – An example of a configuration file expected by OAS DB’s CLI. . . . .	63
Figure 19 – An example of calling OAS DB’s CLI from a terminal. . . . .	63
Figure 20 – Function that generates an invalid example by removing keys from the valid example provided in the OpenAPI seed. . . . .	64
Figure 21 – Function that purposefully fails to delete a record if this is one of the faults the user wished to inject in the mock API. . . . .	65
Figure 22 – UML Component Diagram of Oasis . . . . .	68

## List of Tables

Table 1 – Search strings and last search date per database . . . . .	31
Table 2 – Results per scientific database . . . . .	32
Table 3 – Anti-patterns and the ISO 25010:2011 attributes affected . . . . .	54
Table 4 – Experimental Evaluation Results - Oasis . . . . .	73
Table 5 – Experimental Evaluation Results - RESTler . . . . .	75
Table 6 – Experimental Evaluation Results - RESTest . . . . .	77

## **List of abbreviations and acronyms**

API	Application Programming Interface
CVE	Common Vulnerabilities and Exposures
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
JSON	JavaScript Object Notation
JWT	JSON Web Token
OCCI	Open Cloud Computing Interface
RAML	RESTful API Modeling Language
REST	Representational State Transfer
SCA	Service Component Architecture
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifier

## Contents

<b>1</b>	<b>Introduction . . . . .</b>	<b>15</b>
1.1	<i>Context . . . . .</i>	15
1.2	<i>Motivations . . . . .</i>	15
1.3	<i>Research gap . . . . .</i>	16
1.4	<i>Hypotheses . . . . .</i>	17
1.5	<i>Objectives . . . . .</i>	17
1.6	<i>Research contribution . . . . .</i>	17
1.7	<i>Document structure . . . . .</i>	18
<b>2</b>	<b>Background . . . . .</b>	<b>19</b>
2.1	<i>HTTP . . . . .</i>	19
2.2	<i>SOAP . . . . .</i>	20
2.3	<i>REST . . . . .</i>	21
2.4	<i>Creating an article with REST . . . . .</i>	23
2.5	<i>REST (anti) patterns . . . . .</i>	24
2.6	<i>API and OpenAPI . . . . .</i>	25
2.7	<i>IBM OpenAPI Validator . . . . .</i>	26
2.8	<i>Sinatra . . . . .</i>	26
2.9	<i>Docker . . . . .</i>	27
2.10	<i>Final remarks . . . . .</i>	28
<b>3</b>	<b>Literature review . . . . .</b>	<b>29</b>
3.1	<i>Literature review protocol and objectives . . . . .</i>	29
3.1.1	Selected scientific databases . . . . .	29
3.1.2	Inclusion and exclusion criteria . . . . .	29
3.1.3	Search strings . . . . .	30
3.1.4	Results . . . . .	31
3.1.5	Structure of the remainder of this chapter . . . . .	32
3.2	<i>Recommended good practices for the design of REST APIs . . . . .</i>	32
3.2.1	A study of the design quality of Cloud Computing REST APIs . . . . .	32
3.3	<i>Automatic checking of adherence to REST good practices . . . . .</i>	33

3.3.1	Securing REST APIs through a specification . . . . .	33
3.3.2	Detection of OCCI and REST patterns and anti-patterns . . . . .	35
3.3.3	UniDoSA: Unified Detection of Service Antipatterns . . . . .	36
3.4	<i>Automatic generation of tests from specifications . . . . .</i>	37
3.4.1	Generating test cases from an OpenAPI specification . . . . .	37
3.5	<i>Automatic detection of bugs in REST APIs . . . . .</i>	38
3.5.1	RESTler: a stateful REST API fuzzer . . . . .	38
3.5.2	REStest: automated black-box testing of RESTful Web APIs . . .	39
3.6	<i>Competing OpenAPI repositories and collections . . . . .</i>	40
3.7	<i>Final remarks . . . . .</i>	41
4	<b>REST anti-patterns . . . . .</b>	45
4.1	<i>Proposed new anti-patterns . . . . .</i>	45
4.1.1	Sequential integers as resource ID . . . . .	45
4.2	<i>Discussing an example of a cataloged REST anti-pattern . . . . .</i>	48
4.2.1	Sensitive information in the path or in the query string . . . . .	48
4.3	<i>Final remarks . . . . .</i>	49
5	<b>OAS DB: A generator of annotated specifications and mock APIs to support OpenAPI research . . . . .</b>	50
5.1	<i>OpenAPI specification seed . . . . .</i>	51
5.2	<i>OpenAPI specifications . . . . .</i>	52
5.2.1	REST anti-patterns . . . . .	53
5.3	<i>Mock API implementations . . . . .</i>	55
5.3.1	Code generation . . . . .	55
5.3.2	Fault and issue injection . . . . .	56
5.3.3	In-memory database . . . . .	58
5.3.4	Field validation . . . . .	59
5.4	<i>Annotation files . . . . .</i>	61
5.5	<i>Using OAS DB: from seed to running mock API . . . . .</i>	61
5.5.1	CLI and configuration file . . . . .	62
5.5.2	OAS DB Enhancer Engine . . . . .	63
5.5.3	File Generation . . . . .	65

5.6	<i>Contributing to OAS DB</i> . . . . .	65
5.7	<i>Final remarks</i> . . . . .	66
<b>6</b>	<b>Oasis: a tool for detection of anti-patterns in REST APIs</b> . . .	67
6.1	<i>Overview of Oasis</i> . . . . .	68
6.2	<i>Static analysis</i> . . . . .	69
6.3	<i>Anti-patterns currently detectable</i> . . . . .	69
6.3.1	Anti-pattern Amorphous URI . . . . .	70
6.3.2	Anti-pattern Crudy URI . . . . .	70
6.3.3	Anti-pattern Sensitive Information in the Path or Query String . .	70
6.4	<i>Comparison with other tools</i> . . . . .	70
6.5	<i>Final remarks</i> . . . . .	71
<b>7</b>	<b>Validation</b> . . . . .	72
7.1	<i>Validating OAS DB with tools employing static analysis</i> . . . . .	72
7.1.1	Experiment settings . . . . .	72
7.1.2	Experiment discussion . . . . .	73
7.1.3	Experiment conclusion . . . . .	74
7.2	<i>Validating OAS DB with tools employing dynamic analysis</i> . . . . .	74
7.2.1	RESTler . . . . .	75
7.2.2	RESTest . . . . .	76
7.2.3	Experiment conclusion . . . . .	78
7.3	<i>Novel REST anti-patterns</i> . . . . .	78
7.4	<i>Threats to validity</i> . . . . .	79
7.4.1	External validity . . . . .	79
7.4.2	Internal validity . . . . .	79
7.4.3	Conclusion validity . . . . .	80
7.5	<i>Final remarks</i> . . . . .	80
<b>8</b>	<b>Conclusion</b> . . . . .	81
8.1	<i>Contributions</i> . . . . .	82
8.1.1	Proposal of a new REST anti-pattern . . . . .	82
8.1.2	OAS DB . . . . .	82
8.1.3	Oasis . . . . .	83

8.1.4	Future work . . . . .	83
	Bibliography . . . . .	84
	<b>APPENDIX</b>	<b>87</b>
	APPENDIX A – incident_response.json OpenAPI sample seed	88
	APPENDIX B – payment.json OpenAPI sample seed . . . . .	90
	APPENDIX C – project_management.json OpenAPI sample seed . . . . .	92
	APPENDIX D – Anti-patterns and issues that OAS DB is ca- pable of injecting during asset generation .	94

## 1 Introduction

### 1.1 Context

A popular choice when building Web systems and APIs is to use the REST (Representational State Transfer) architectural style. Introduced in 2000 (FIELDING, 2000), 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, since many of the available tools are adaptations of solutions created to handle older architectures (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019). 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 analyses.

A further challenge faced by the 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.

### 1.2 Motivations

It is common knowledge the great success achieved by the Web in the last decades. More recently, an approach that gained popularity in the Web community is the usage of IaaS (Infrastructure as a Service) solutions, such as Google Cloud Platform and Amazon Web Services (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019). These IaaS solutions allow one to access hardware resources on demand, which is a great change from the recent past when companies had to acquire hardware before even launching a Web software product. This development reduced market entry barriers and, as a consequence, there was a significant increase in companies launching Web products, including services offering Web APIs.



The increase in the use of REST Web APIs does not seem to have had a significant impact on the practices used to check their quality. Automatic verification for compliance with best practices is not yet widespread (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019). As previously explained, the scarcity of tools developed from the ground up specifically to be used with REST APIs may be the main cause for that (i.e., existing tools are not considered good enough). One of the reasons contributing for a slower progress in the research and development of new tools may be the lack of comprehensive and shared datasets, which could be leveraged by researchers and practitioners while developing new tools and techniques. Considering all the aforementioned facts, it appears to be important to invest in the research and development of datasets, techniques and tools to be specifically used with REST Web APIs and by its researchers.

### 1.3 Research gap

As mentioned in previous sections, companies and the Web community in general are using specification languages — such as OpenAPI and RAML<sup>1</sup> (RESTful API Modeling Language) — to describe REST Web APIs. Despite the usage of these specifications, it is not yet common to leverage all the details present in the specifications to automatically check for compliance to REST best practices or to detect common pitfalls; nonetheless, the relevant literature does provide compilations of these best practices and pitfalls (such as Petrillo *et al.* (2016)). This is an underexplored opportunity for detecting potential issues in these APIs. Besides that, one notices that researchers in this field are in general using their own custom datasets. This is detrimental because it means time is diverted from the main purposes of each research in order to build datasets and it also makes the comparison between studies challenging.

OAS DB — our main contribution — aims to address this gap by generating specifications and their corresponding mock implementations. These can have issues and anti-patterns injected into them. The issues and anti-patterns that OAS DB is able to introduce in these generated assets are supported by the relevant literature. Chapter 5 explores all these and other related topics in depth.

---

<sup>1</sup> <https://raml.org/>

## 1.4 Hypotheses

We put forward the hypothesis that a tool that can generate complete OpenAPI specifications and their corresponding mock API implementations could be a useful resource for researchers and practitioners in the REST API/OpenAPI realm. Since this is a novel contribution in the OpenAPI realm, no direct comparison with existing solutions was possible.

To validate that our main contribution (OAS DB) can generate useful specification/implementation pairs, we ran three experiments showing that it is not only able to generate assets with issues backed up by the literature but also that it is capable of generating specification/implementation pairs with issues and anti-patterns not detected by existing tools. Each experiment used a different tool. Chapter 7 explains and discuss in-depth these experiments.

## 1.5 Objectives

This study had one main objective: to create a tool capable of generating synthetic but realistic OpenAPI samples. Together with the sample, the tool is also able to generate an annotation file and the corresponding mock API implementation.

The annotation file describes which anti-patterns/issues the specification/mock API contain and the segment in the specification associated with the problem. The mock API implementation can have issues that are not directly specifiable in the OpenAPI sample and showed to be a resource of great value to test tools that use dynamic analysis techniques.

## 1.6 Research contribution

We expect that OAS DB — and its pioneering approach — will help researchers in the field. We expect as main benefits a) cost saving, b) the accelerated improvement of datasets (since researchers from different teams are able to collaborate and help evolve OAS DB) and c) easier comparison between different tools that use OAS DB for validation.

Positive effects as such have already been demonstrated in other fields, as is shown for example by Do, Elbaum and Rothermel (2005) and Just, Jalali and Ernst (2014).

In particular, our study resulted in the following products:

- A comprehensive solution (OAS DB) for generating OpenAPI specifications and their corresponding mock API implementations and annotation files.
- A novel anti-pattern whose avoidance result in increased security of Web REST APIs (and of the business data that they operate);
- A proof of concept tool for detection of REST anti-patterns (Oasis) on OpenAPI specifications by means of static analysis.

### *1.7 Document structure*

The remainder of this document is structured as follows: Chapter 2 presents all the background knowledge considered sufficient for better understanding in which context this research is immersed in; Chapter 3 explores and examines the most relevant related works; Chapter 4 proposes novel REST anti-patterns; Chapter 5 introduces OAS DB, a tool for generating OpenAPI samples containing known anti-patterns and issues; Chapter 6 presents Oasis, a proof of concept tool for detecting anti-patterns on OpenAPI specifications; Chapter 7 goes through all the approaches and experiments used to validate the products of our research; finally, Chapter 8 closes this document by summarizing the research results and by briefly discussing some possibilities for future work.

## 2 Background

This chapter presents the most important concepts and tools relevant to this work. We will explore core Web technologies and architectures (such as HTTP and REST), essential concepts (such as what constitutes a REST anti-pattern) and also tools that are used in our research (such as the IBM OpenAPI Validator).

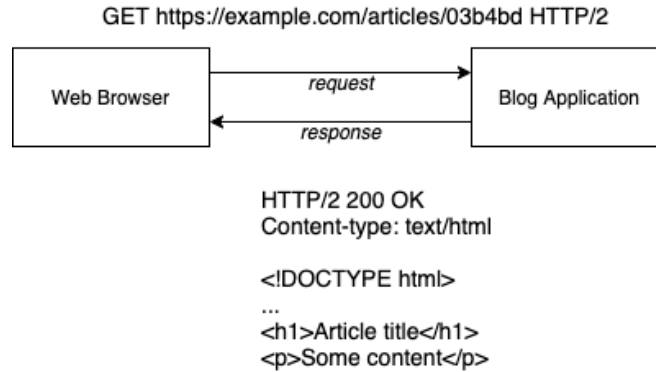
### 2.1 HTTP

HTTP stands for Hypertext Transfer Protocol. It is an application level and a stateless protocol (RFC 7231, 2014). It underpins the World Wide Web and is used every time one accesses a website or interacts with a Web application.

Communication between a client and a server using the HTTP protocol happens in the following fashion: the client constructs and issues a request; the server parses and interprets the message and responds with one or more messages; finally, the client analyzes the response and determines if the requested action was carried out successfully or not by the server (RFC 7231, 2014). Figure 1 illustrates in a simplified form the HTTP messages exchanged by a Web browser and a blog application when a user requests a given article.

HTTPS (Hypertext Transfer Protocol Secure) is the term used when HTTP is used over TLS (Transport Layer Security), a secure communication protocol (RFC 2818, 2000). TLS guarantees communication will have the following properties: authentication (at least the server — and optionally the client as well — proves its identity), confidentiality (only the peers communicating can view the data being exchanged) and integrity (data cannot be modified by attackers without detection) (RFC 8446, 2018).

Figure 1 – Interaction with an example REST Blog Application: the user requests (by using the Web Browser) a specific article; the server responds the GET HTTP request with the contents of the article in the HTML format (which will then be rendered into a user interface by the Web Browser)



Source: Alex Braha Stoll, 2022

## 2.2 SOAP

SOAP (Simple Object Access Protocol) is a messaging protocol that allows communication among different web services in a computer network. Messages are exchanged in the XML (Extensible Markup Language) format. It relies on application level protocols, most commonly on HTTP. However, since SOAP is application protocol agnostic, it is possible to utilize SOAP over an ESB (Enterprise Service Bus) or even SMTP (Simple Mail Transfer Protocol) (SOAP Specification, 2007).

Figure 2 shows an example SOAP message. Every SOAP message is contained within an *envelope* element. The *header* piece can contain information useful to intermediaries or even the final message destination. The *priority* field inside the *header* element, for example, could be used by an intermediary when deciding which messages to relay first. Finally, the *body* element contains the message payload (SOAP Specification, 2007).

Figure 2 – An example of SOAP message

```

1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
2    <env:Header>
3      <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
4        <n:priority>1</n:priority>
5        <n:expires>2001-06-22T14:00:00-05:00</n:expires>
6      </n:alertcontrol>
7    </env:Header>
8    <env:Body>
9      <m:alert xmlns:m="http://example.org/alert">
10       <m:msg>Pick up Mary at school at 2pm</m:msg>
11     </m:alert>
12   </env:Body>
13 </env:Envelope>

```

Source: Alex Braha Stoll, 2022

The first draft of SOAP 1.2 (the latest available version) dates back to July of 2001 (SOAP Draft, 2001). Nowadays, it is a technology with declining popularity. It is still used in some domains where high security, complex transactions and support for legacy systems is of utmost importance, such as banking systems (Official Raygun Blog, 2020). In general, however, other solutions for organizing and interacting with web services are now preferred over SOAP, such as REST (Representational State Transfer), discussed in section 2.3.

### 2.3 REST

REST (Representational State Transfer) is an architecture style for Web services. Its defining characteristic is allowing the manipulation of Web resources by the usage of a set of predefined operations in an stateless manner. REST aims at improving the scalability of interaction between components, their independent deployment, and the generality of interfaces. REST also has the objective of allowing intermediary components to reduce the latency of interactions, to enforce security constraints and to also be able to encapsulate legacy systems (FIELDING, 2000).

To see many of these features in practice, let us go through possible interactions with a fictitious Web service. Our example application is a blog that offers two operations: to read and to write an article. Since our application follows the REST architecture, each article is identified by an URI (Uniform Resource Identifier). The URI for an article is `https://example.com/articles/ARTICLE_ID`, in which *ARTICLE\_ID* is the non-

sequential unique identifier of a given article (e.g., 03b4bd). As the article URI implies, our service is powered by HTTPS (the secure version of the HTTP protocol). To return the current state of a given resource, a REST compliant system must expect a GET request to the URI of the resource (RFC 7231, 2014). To read a given article, then, we would issue a GET HTTP request and — assuming we are doing so by using a Web Browser — the server would respond with the desired Web resource (the article we intend to read) represented in the HTML (Hypertext Markup Language) media type. Figure 1 illustrates — in a simplified form — this very operation.

As explained in Section 2.1, HTTP is stateless. Adding this fact with the read-only semantics of the HTTP GET verb (RFC 7231, 2014), the example illustrated in Figure 1 allows us to demonstrate many of the main positive characteristics of a REST system: its client-server architecture, its statelessness, its cacheability and its potential for being a layered system (FIELDING, 2000):

1. **Client-server architecture:** the server is responsible for responding with a representation of the resource the client asked for; the client (Web Browser) is the one responsible for rendering a user-interface from this representation. This separation allows both parts of the system to evolve independently.
2. **Statelessness:** HTTP is stateless and an HTTP request therefore carries all the information necessary for it to be processed by the server. As a consequence of that, if we were to request the same article once again we would retrieve the exact same response, even if the server was restarted between the two requests (of course assuming no other request changed the state of the article itself).
3. **Cacheability:** because of what is stated by item 2, requests to read an article could be cached and then to serve the client it would not be necessary to every time do the same amount of processing that was done when the request was first made (as long as the cache remains valid).
4. **Layered system:** since the client (Web Browser) is totally decoupled from the server (as explained in item 1), the server-side component of our system can be composed of multiple subsystems. To implement a cache for the articles resource (as described in item 3), we could for example use an HTTP-level caching mechanism such as Varnish<sup>1</sup>.

---

<sup>1</sup> <https://varnish-cache.org/>

## 2.4 Creating an article with REST

Let us now explore how our REST blog application implements the creation of articles. REST allows an application to accept and respond with representations of a Web resource in different media types (FIELDING, 2000). To show this in practice, let us assume that our blog application has an API (Application Programming Interface) including an operation to create new articles. Let us suppose that we have a mobile application that empowers users to create articles on the go. When receiving a request for the creation of a new article, this time our application will expect the Web resource (the article) represented as JSON (JavaScript Object Notation), which is a lightweight data-interchange format based on a subset of the JavaScript programming language<sup>2</sup>. The response will also be in this same media type.

The action of creating a new article is one that causes side effects on the server (the creation of a new record, the article itself) and therefore is not idempotent<sup>3</sup>. The REST architecture requires HTTP verb semantics to be respected (FIELDING, 2000); the verb that has a semantic compatible with this scenario is POST (RFC 7231, 2014). Figure 3 illustrates the interaction that happens between the mobile application and the blog application for the creation of a new article.

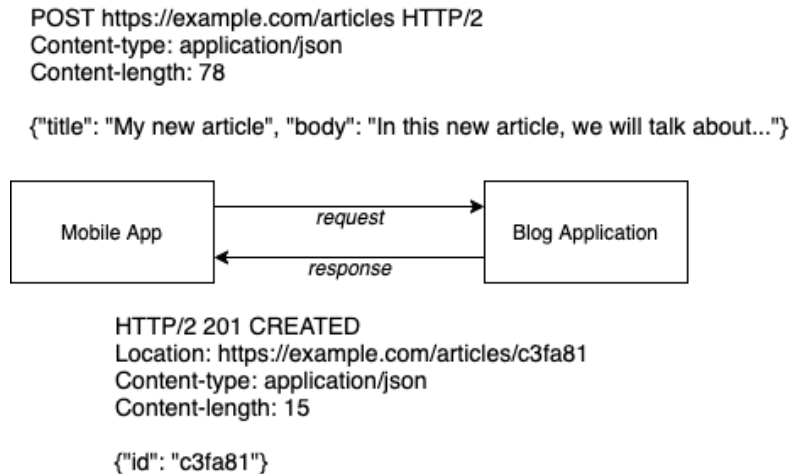
---

<sup>2</sup> <https://www.json.org>

<sup>3</sup> The term *idempotent* is being used here as it is in the jargon of REST applications. An idempotent operation is one that does not cause side effects on the server (e.g., the creation of a record would be a side effect) and can be safely repeated and cached.



Figure 3 – By using the mobile application to create a new article, a POST request with the contents of the article represented as JSON is issued to the server. Note that the server not only responds with a status code indicating success, but also includes the URI of the newly created Web resource (the new article) in the Location header



Source: Alex Braha Stoll, 2022

A final pillar of REST that this last example illustrates is the role of hypermedia as the engine of the application state (FIELDING, 2000). Figure 3 shows that the server includes in the HTTP Location header of the response the URI for the newly created article. Note that this is the same URI — of course with a different article ID — that appears when we were discussing the implementation of the feature of reading an article.

This role of hypermedia in REST applications is beneficial because it allows the client-side (be it a mobile application, be it a browser-based UI etc) to explore resources without (or at least with less) implementation overhead. That is the case because the client-side of a REST compliant system can assume for every different resource creation endpoint that the URI of the newly created object will be available in the HTTP Location header of the server response. For applications that do not follow REST, that is probably not the case and even the way in which one should access different resources may be inconsistent across the API, probably resulting in greater implementation effort.

## 2.5 REST (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 (BRABRA *et al.*, 2019). 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.* (2019)).

In Section 2.3, we saw that the appropriate HTTP verb for reading an article is GET. If a system that claims to abide by the REST architecture uses the POST verb for such kinds of actions, this would be an example of a REST anti-pattern. POST is to be used for actions that cause side effects (RFC 7231, 2014) and a consequence of using it in idempotent actions is wrongfully preventing the caching of the interaction (intermediary network nodes between the client and the server may use information such as the HTTP verb used to decide if it is safe to cache an operation). REST anti-patterns will be discussed in much more detail in Chapter 4.

## 2.6 API and OpenAPI

Application Programming Interface (API) is a specified set of operations for programmatically interacting with components of a software system. In particular, an Web API is an interface that allows an web system to receive requests from other systems (MAXIMILIEN; RANABAHU; GOMADAM, 2008).

OpenAPI<sup>4</sup> is a specification to describe all the resources that can be accessed and manipulated through a REST Web API. An OpenAPI specification offers a level of detail sufficient for different use cases, such as generation of clients able to interact with the API and generation of mock servers able to respond to real requests. Figure 4 shows a segment of an OpenAPI specification. This snippet describes an operation that can be requested through the `/orders` path (see line 2 of Figure 4), that expects the HTTP verb GET to be used (line 3) and when successful that responds with a collection of orders (line 14).

---

<sup>4</sup> <https://www.openapis.org/>

Figure 4 – A segment of an OpenAPI specification documenting a particular operation available in the API being described

```

1  paths:
2    /orders:
3      get:
4        summary: List all orders of the authenticated user.
5        operationId: list_orders
6        tags:
7          - orders
8        responses:
9          200:
10         description: An array of orders
11         content:
12           application/json:
13             schema:
14               $ref: "#/components/schemas/Orders"

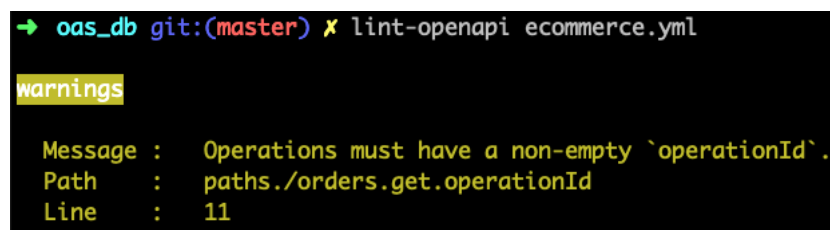
```

Source: Alex Braha Stoll, 2022

## 2.7 IBM OpenAPI Validator

IBM OpenAPI Validator<sup>5</sup> is an open-source command line tool that allows one to check an OpenAPI specification (version 2 or higher) and look for violations of OpenAPI good practices. An example of an offence would be not to fill a recommended field, as shown in Figure 5.

Figure 5 – IBM OpenAPI Validator reporting an offence after linting an OpenAPI specification



```

→ oas_db git:(master) ✗ lint-openapi ecommerce.yml

warnings

Message : Operations must have a non-empty `operationId`.
Path    : paths./orders.get.operationId
Line    : 11

```

Source: Alex Braha Stoll, 2022

## 2.8 Sinatra

Sinatra<sup>6</sup> is an open-source web framework for the Ruby programming language. By using the framework, it is possible not only to create single-file, self-contained web systems

<sup>5</sup> <https://github.com/IBM/openapi-validator>

<sup>6</sup> <https://www.sinatrarb.com>

and web APIs, but also much larger systems. The framework is very straightforward to use and because of that it lends itself well for some special cases, like for example automatically generating the code for a web system or API.

To see in practice how one would build a simple API with Sinatra, let us go through an example. Let us say that we wanted to build a very simple service that received a GET HTTP request at a `/hi` endpoint and returned a `text/plain` response with the string “Hello, World!”. Below, Figure 6 shows all the code necessary to implement this trivial API.

Figure 6 – A simple API implemented by using the Sinatra framework, available for the Ruby programming language.

```

1  require 'sinatra'
2
3  get '/hi' do
4    'Hello, World!'
5  end

```

Source: Alex Braha Stoll, 2022

## 2.9 Docker

Docker<sup>7</sup> is a toolchain powering containerized software. A container can be understood as a unit of software isolated from the rest of the environment where it is running on (Official Docker Website, 2020).

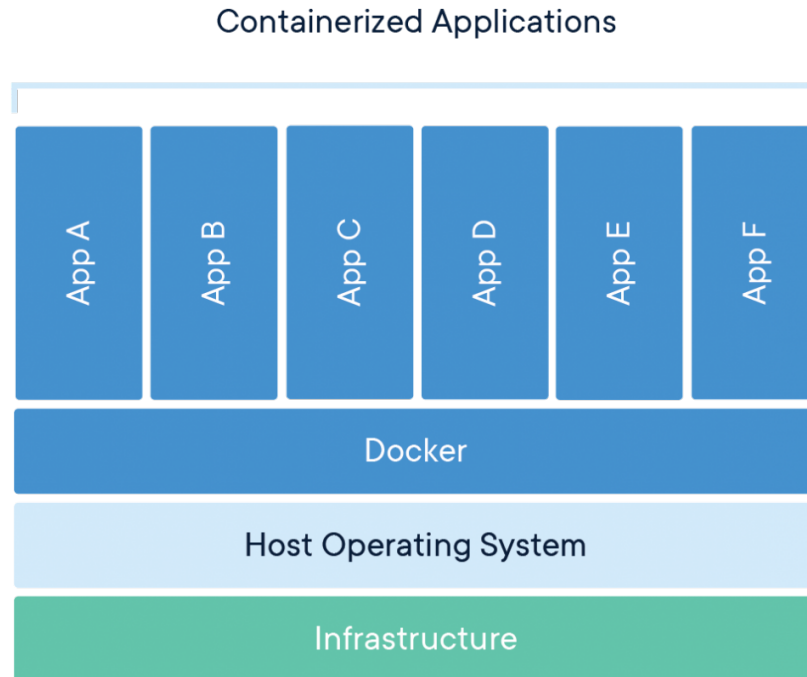
Programs — let us say a Web application — normally depend on other software being available in the environment they are running on (for example, a Linux server). This can cause issues if one needs to deploy a given program in a new environment and if by mistake not all dependencies are installed and configured in this new target. If a container is created including all the dependencies of a given program, it is safe to assume that the software will work as expected even in a new environment (Official Docker Website, 2020).

Multiple containers can run in the same infrastructure sharing its kernel (let us say multiple Web applications running on a Linux server), making it possible to better utilize the same hardware in a safe manner (because as explained containers are designed to be isolated from one another and cannot arbitrarily interfere in each other) (Official Docker Website, 2020). Figure 7 illustrates these concepts.

---

<sup>7</sup> <https://www.docker.com>

Figure 7 – Multiple containerized applications, each with its own isolated environment but still sharing the operating system kernel via Docker



Source: Official Docker Website (2020)

### 2.10 Final remarks

In this chapter, we presented the essential background knowledge upon which our research rests. We covered protocols (HTTP and SOAP), architectural styles (REST) and also concepts and specifications (API and OpenAPI, respectively). We also presented technologies that are incorporated into the products of our research, such as the Web framework Sinatra.

### 3 Literature review

#### 3.1 *Literature review protocol and objectives*

To get familiar with the state-of-the-art, we conducted a search through the main databases relevant to our field. Then, after analyzing the title, abstract and keywords of the papers that emerged, we selected only those that satisfied all of the inclusion criteria (and, conversely, did **not** satisfy any of the exclusion criteria). The papers that were chosen were read in full and the most important ones have dedicated subsections in this literature review.

The review aims to answer to the following questions in particular: 1) Which specifications — if any — are most commonly used by researchers when developing techniques and tools for REST APIs?; 2) Are the issues commonly found in REST APIs fully mapped by the research community?; and 3) Which datasets are used by researchers in this field?

##### 3.1.1 Selected scientific databases

The main databases relevant to our field of research — and those in which we conducted a search — are the following:

- IEEE Xplore
- ACM Digital Library
- SCOPUS

##### 3.1.2 Inclusion and exclusion criteria

To be selected, a paper must satisfy **all** of the inclusion criteria and it must **not** satisfy any of the exclusion criteria. Both criteria are listed below.

##### Inclusion criteria

- The authors of this review must have access to the full version of the study;

- Only studies published after 2010 were appraised. This year was chosen as the starting point because it is the creation year of a popular REST API specification language (OpenAPI);
- Only studies that propose techniques for evaluating the quality or security of an API or introduce techniques for automatically generating tests for an API will be considered;
- To be considered, studies must involve APIs that follow the REST architectural style;
- Evaluation or test generation must be done by leveraging some type of specification of the API;
- The study must be published in a journal, conference or symposium directly related with Computer Science.

#### Exclusion criteria

- The authors of this review did not have access to the full version of the study;
- Studies published before 2010 were not considered;
- Works that do not propose techniques for evaluating the quality or security of an API or introduce techniques for automatically generating tests for an API will not be included;
- Studies that involve APIs that do not follow the REST architectural style will be ignored;
- Studies that do not use some kind of API specification will not be considered;
- Works that are not published in a journal, conference or symposium directly related to Computer Science will be ignored.

#### 3.1.3 Search strings

Table 1 shows the search string used in each database and the date when the latest search was performed. The search strings are all equivalent. Their variation is due to the different search commands / interfaces supported by each database.

Table 1 – Search strings and last search date per database

Database	Search date	Search String
ACM Digital Library	May 2nd, 2022	Title:((“rest api” AND spec) OR (“rest api” AND specification) OR (“restful api” AND spec) OR (“restful api” AND specification)) OR Abstract:((“rest api” AND spec) OR (“rest api” AND specification) OR (“restful api” AND spec) OR (“restful api” AND specification)) OR Keyword:((“rest api” AND spec) OR (“rest api” AND specification) OR (“restful api” AND spec) OR (“restful api” AND specification))
IEEE Xplore	May 2nd, 2022	((“All Metadata”:“rest api” AND spec) OR (“All Metadata”:“rest api” AND specification) OR (“All Metadata”:“restful api” AND spec) OR (“All Metadata”:“restful api” AND specification))
SCOPUS	May 2nd, 2022	TITLE-ABS-KEY ( “rest api” AND spec ) OR TITLE-ABS-KEY ( “rest api” AND specification ) OR TITLE-ABS-KEY ( “restful api” AND spec ) OR TITLE-ABS-KEY ( “restful api” AND specification )

Source: Alex Braha Stoll, 2022

### 3.1.4 Results

As a result of the review process described in previous sections, a total of 17 unique works were selected (14 as a result of searching scientific databases and three already known by the authors of this study). Across all selected scientific databases, a total of 147 studies were retrieved. For details on the results obtained for each database, see Table 2.



Table 2 – Results per scientific database

Database	Total retrieved	Total selected
ACM Digital Library	12	3
IEEE Xplore	37	7
SCOPUS	98	13

Source: Alex Braha Stoll, 2022

### 3.1.5 Structure of the remainder of this chapter

The rest of this chapter is dedicated to presenting summaries of the most relevant among the papers that satisfied the inclusion criteria aforementioned. The summaries are grouped by theme (e.g., *Generating test cases from an OpenAPI specification*). After that, the chapter closes with a discussion on the findings and how they support the proposal here being put forward.

## 3.2 Recommended good practices for the design of REST APIs

### 3.2.1 A study of the design quality of Cloud Computing REST APIs

Cloud Computing has transformed the Information Technology industry (PETRILLO *et al.*, 2016). Cloud Computing providers often also offer their services through APIs and, although there are no widely accepted standard for developing these APIs, it is safe to say that the REST architectural style is a *de facto* standard. Among these REST APIs, however, there is a lot of differences. Therefore, it is difficult to assess the quality of each API.

In order to help facilitate this task, it is proposed a catalog with 73 best practices to be followed when designing an API to enhance its understandability and its reusability. The catalog was built by surveying the literature on design best practices for REST APIs. Besides the catalog, the second important contribution is an analysis of three important APIs for Cloud Computing: Google Cloud Platform<sup>1</sup>, OpenStack<sup>2</sup> and Open Cloud Computing Interface (OCCI)<sup>3</sup>. These APIs were selected because each one of

---

<sup>1</sup> <https://cloud.google.com/>

<sup>2</sup> <https://www.openstack.org/>

<sup>3</sup> <http://occi-wg.org/>

them represent, in order, a commercial offer, an open-source implementation and an open standard (PETRILLO *et al.*, 2016).

In addition to adherence to the cataloged best practices, it was also verified which common categories of operations (in a cloud environment) were supported by the APIs (e.g., virtual machine management, container management, access control etc). All analyses were done manually by the authors. Regarding adherence to REST API design best practices, the three APIs follow an average of 44 out of the 73 proposed practices. Specifically, the results are the following:

- Google Cloud Platform: 48 / 73 (66%)
- OpenStack: 45 / 73 (62%)
- OCCI: 41 / 73 (56%)

As part of the research questions proposed, the authors also identified the best practices that are followed by all APIs and those that are followed by none. It may be interesting to further investigate why that is the case and — concerning practices not being followed by any of the analyzed APIs — to question if they really should belong to a best practices catalog.

The authors conclude that the suggested catalog of best practices help when assessing the design of REST APIs (concerning their understandability and reusability). They also conclude that cloud computing APIs do reach an acceptable level of maturity (again, taking into consideration understandability and reusability), since all of them honor more than 50% of the best practices.

### 3.3 Automatic checking of adherence to REST good practices

#### 3.3.1 Securing REST APIs through a specification

Iversen (2018) shows that it is possible to find security vulnerabilities in REST APIs by leveraging the information contained in a specification. To detect possible security issues, two strategies are employed: static analysis of specifications and generation of tests to be run against APIs by using the information available in their corresponding specifications. The study presents a couple of different detectable security issues, focusing on vulnerabilities associated with the JSON Web Token (JWT) technology. JWT is an

open standard that leverages digital signature algorithms (e.g., public-key cryptography) to allow authentication and secure information sharing between parties across a network (RFC 7519, 2015).

Iversen (2018) introduces a new API specification instead of leveraging one of the specifications used in the industry (e.g., OpenAPI<sup>4</sup>). The decision not to use an established specification is justified by two arguments: 1) by introducing a new specification, there is no need to dedicate resources into dealing with irrelevant (to the purposes of the study) implementation details of one of the industry used specifications; 2) being able to express authorization constraints (i.e., user access levels).

The static analysis strategy uses two techniques. The first one is to employ dictionaries with patterns to be searched for in the specification. One example of a vulnerability detectable by using this technique is having a mandatory parameter (e.g., a user identification) to be present in the query string of an endpoint URI (Uniform Resource Identifier) instead of being present as a URI parameter. The second technique utilized makes use of algorithms that are able to do some kind of verification against available information in order to produce a reliable result about the presence or absence of a given problem (e.g., specifying “none” as the hashing algorithm to be used when signing a JWT token). The first technique may result in false positives (because not every detection of a dictionary pattern necessarily means an issue), while the second cannot produce false positives (the nature of the utilized algorithms always detect issues when they are present) (IVERSEN, 2018).

The dynamic analysis strategy leverages the API specification to make requests against the API. By making real requests, it is shown that it is possible to detect a different set of security vulnerabilities. The detected issues can be categorized as following: 1) comparison of response HTTP codes with expected codes; 2) verification of the returned payload against the expected return fields (i.e., verifying if the API is possibly leaking data); and 3) to check if it is possible to write to fields that are not supposed to be mutable (for a given user access level). It is also proposed the combined usage of all the dynamic techniques in stages, starting with the verification of HTTP codes and then proceeding to the next techniques whenever the currently executing succeeds. It is also proposed that checks should be executed for every user access level and endpoint.

---

<sup>4</sup> <https://www.openapis.org/>

### 3.3.2 Detection of OCCI and REST patterns and anti-patterns

Brabra *et al.* (2019) stated that the pay-as-you-go and elasticity characteristics of Cloud Computing are contributing to it becoming more and more attractive to software teams. There are many different providers of this type of solution (e.g., Amazon Web Services<sup>5</sup> and Google Cloud Platform<sup>6</sup>), making interoperability a challenge since each provider offers a different API. An initiative to facilitate interoperability among these services is the Open Cloud Computing Interface (OCCI), an open standard that aims to provide a meta-model for managing cloud resources (i.e., computing, storage and other services available through a platform like Amazon Web Services). The OCCI standard also specifies a REST API through which one can interact with the aforementioned cloud resources.

There are tools to help developers check whether the APIs they are developing conform to OCCI best principles and to REST best practices, however these tools have shortcomings. As an example, there is an official OCCI tool<sup>7</sup> that allows one to detect best practices, however this utility does not show detailed descriptions of the practices that it detected being followed, nor lists the practices that it did not detect as being honored. Brabra *et al.* (2019) proposes an approach that not only automatically detect OCCI and REST patterns and anti-patterns, but also offers a set of correction suggestions when anti-patterns are identified. In order to do so, the proposed technique / proof of concept tool relies on semantic models manually built from the documentation of a cloud provider API.

To verify the proposed technique, the researchers decided to run a proof of concept tool against five cloud APIs: OOi, COAPS<sup>8</sup>, OpenNebula<sup>9</sup>, Amazon S3 and Rackspace<sup>10</sup>. Besides manually building semantic models for these APIs (as mentioned before), the APIs were also manually analyzed to check compliance for the same set of REST and OCCI patterns that are checked by the developed tool. Compliance (to REST or OCCI) was defined as the percentage of patterns that each API operation conforms to in relation to

---

<sup>5</sup> <https://aws.amazon.com/>

<sup>6</sup> <https://cloud.google.com/>

<sup>7</sup> OCCI Compliance Testing Tool

<sup>8</sup> <http://www-inf.int-evry.fr/SIMBAD/tools/COAPS1/>

<sup>9</sup> <https://openebula.org/>

<sup>10</sup> <https://www.rackspace.com/cloud>

all the patterns that are applicable (i.e., it would not make sense to check if an operation that expects the GET HTTP verb conforms to a good practice specific to the POST verb).

The results of these experiments showed that all the APIs aforementioned already reached an acceptable REST compliance degree. The mean value for the studied APIs was greater than 50%. Regarding compliance with OCCI best practices, the results were inferior. No API reached a compliance greater than 58% and the least compliant API (Amazon S3) scored a compliance measure of only 42%. This shows that although these APIs have achieved good REST compliance, the developers of each solution are not yet giving a lot of importance to following OCCI best principles.

### 3.3.3 UniDoSA: Unified Detection of Service Antipatterns

Palma, Moha and Guéhéneuc (2018) presents a tool (UniDoSa) capable of detecting anti-patterns in three different web service technologies: Representational State Transfer (REST), Service Component Architecture (SCA) and Simple Object Access Protocol (SOAP). To accomplish that, the authors introduce: a) a meta-model in a higher level of abstraction capable of representing and relating the concepts in the aforementioned web service technologies; b) a Domain-specific language (DSL) for describing anti-patterns in terms of static and dynamic properties observed in the services under analysis; and c) a framework — called Service Oriented Framework for Antipatterns (SOFA) — that supports the process of collection of metrics from the service under analysis (to be used to determine if a given anti-pattern is present or not) and the process of automatic generation of executable code for detection of anti-patterns (based on rule cards describing each anti-pattern in the DSL aforementioned).

Part of the technology behind UniDoSa — the detection of REST anti-patterns using static and dynamic techniques — is also available as a web tool called WebRestpad<sup>11</sup>. The tool allows one to scan a REST API, being able to detect six REST anti-patterns using static techniques and eight different ones when selecting the option to use dynamic techniques.

To validate UniDoSA, an experiment was run to attempt to detect different service anti-patterns selected from the literature in a collection of web services in the three technologies supported by the tool. Specifically, a total of 12 different anti-patterns were

---

<sup>11</sup> <http://webrestpad.sofa.uqam.ca/>

searched for in 18 REST APIs, 2 SCA systems and 120 SOAP services (it is important to mention that not all those 12 anti-patterns are relevant to all technologies). The detections made by UniDoSA were then manually validated by the research team. In this experiment, UniDoSA showed an average precision of 89.78% and a recall of 96.67%.

As future work, the authors intend to investigate the potentially negative impacts the anti-patterns detected may cause in the systems under study. They also plan to work in the correction of the detected anti-patterns by using semi-automated approaches.

### 3.4 *Automatic generation of tests from specifications*

#### 3.4.1 Generating test cases from an OpenAPI specification

An OpenAPI specification allows one to describe the details of the interface of an API following the REST architectural style. Different than other specifications, however, the OpenAPI standard was designed specifically with the objective of describing REST APIs. Other specifications, like the Web Application Description Language<sup>12</sup>, were proposed to describe REST APIs, but failed due to its complexity and inadequacy in describing this particular architectural style (ED-DOUIBI; IZQUIERDO; CABOT, 2018).

Regarding the automatic generation of test cases from a specification, many approaches have been proposed, but mostly for APIs using SOAP (Simple Object Access Protocol). Although it is possible to design a REST API that uses the Simple Object Access Protocol, APIs following REST generally use the Hypertext Transfer Protocol (HTTP). There are also proposals and tools to generate test cases from REST compatible specifications, but they generally require the developer to input data or do not provide support for testing fault-based scenarios (ED-DOUIBI; IZQUIERDO; CABOT, 2018). Besides that, some approaches introduce a custom specification for describing the API or require one to have access to the source code (e.g., Iversen (2018)).

Ed-Douibi, Izquierdo and Cabot (2018) proposes a technique and a tool that is able to generate test cases for REST APIs having as the sole requirement a valid OpenAPI specification describing the API that one wants to test. A developer does not need to input any other data and the tool is also able to test fault-based scenarios in which invalid data is willingly sent to check if the service correctly responds with the expected error code.

---

<sup>12</sup> <https://www.w3.org/Submission/wadl/>

The technique for generating the test cases is composed of four steps. First, an OpenAPI model is built from the specification. This is followed by the generation of input data. Then, test suite models are created. Finally, in the fourth and final step, a program is generated to be run against the API and do the actual testing.

The verification of the aforementioned tool was done by using it to scan 91 APIs from an open repository of Open API specifications<sup>13</sup>. After generating and running tests for these APIs, it was found that 40% had failed to pass at least one test from the automatically built test suite. The developed proof of concept tool was able to detect errors in many of the tested APIs. However, the authors themselves list some drawbacks that they intend to address in the future. One example is the lack of support for scenarios in which exists a dependency between requests to the API (i.e., some operation can only be tested if another operation is executed previously to first create data or change the state of the system under test). Another example is the lack of support for version 3 of the OpenAPI specification (the tool only supports version 2).

### 3.5 *Automatic detection of bugs in REST APIs*

#### 3.5.1 RESTler: a stateful REST API fuzzer

RESTler is the first stateful API fuzzer (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019). It is able to automatically generate code to fuzz a REST API by analyzing its OpenAPI specification. RESTler has two distinguishing features: a) its ability to infer dependencies among requests (e.g., request A creates a resource that is needed in order to execute request B, therefore B depends on A); and b) RESTler analyzes the response from each request and eliminates invalid sequences of requests (i.e., if the server responds with certain types of error after a sequence A-B is executed, RESTler will no longer attempt this same sequence in the future).

RESTler reads an OpenAPI specification and then generates Python code that is able to make requests to the API. For requests that expect parameters, RESTler is able to expand the original request into multiple ones by drawing different values for each parameter from a dictionary (which can be enhanced by the user). By default, RESTler uses a breadth-first search (BFS) strategy to explore the space of all possible requests

---

<sup>13</sup> <http://apis.guru>

(and sequences thereof). Other algorithms can be used and the authors show experiments also using BFS-Fast and RandomWalk.

The authors ran three experiments with RESTler. One against a simple Blog API, one against GitLab<sup>14</sup> (a popular hosted Git open-source solution) and one against Azure<sup>15</sup> and Office365<sup>16</sup> services. They were able to provide evidence that RESTler capabilities (inferring dependencies and analyzing responses, as explained before) are relevant in enhancing the efficacy of the tool and its ability to exercise more lines of code of the API under test. They were also able to show that RESTler seems to be capable of finding important bugs. In total, 22 were found in GitLab. The authors did not disclose the number of bugs found in Azure and Office365 services, but reported that multiple were found as well.

As future work, the authors propose a series of potential improvements to RESTler. One is to support even richer user annotations on an OpenAPI specification to allow one to specify complex service-specific types. Another possible area of research is how to automatically generate an OpenAPI specification (by using machine learning and traffic logs) for web services that do not offer one. The authors also mention the possibility of adding to RESTler the ability of detecting server-side assertion violations by analyzing back-end logs. The authors conclude the paper by emphasizing the importance of further exploration in the area of automatic detection of bugs in REST APIs, since it is still not completely clear the frequency and categories of issues generally found in these types of web services.

### 3.5.2 RESTest: automated black-box testing of RESTful Web APIs

RESTest is a tool able to generate test cases for REST APIs from their corresponding OpenAPI specification (MARTIN-LOPEZ; SEGURA; RUIZ-CORTÉS, 2021). It supports the following testing techniques: fuzzing, adaptative random testing and constraint-based testing.

RESTest’s workflow consists of the following steps:

---

<sup>14</sup> <https://about.gitlab.com/>

<sup>15</sup> <https://azure.microsoft.com/>

<sup>16</sup> <https://www.office.com/>



1. **Test model generation:** to generate abstract test cases, it uses the API's OpenAPI specification and a configurable YAML file. This configurable file can be edited to include information such as an authentication token to interact with the API. It is also possible to specify particular data generators to be used with each parameters (and the generators themselves can be created by the user);
2. **Abstract test case generation:** Plataform-independent test cases are generated from the models created in the preceding step;
3. **Test case generation:** The abstract test cases are instantiated into executable tests in a particular language or framework. The user can also extend RESTest and create its own test case generators.
4. **Test case execution:** Test cases can then be executed. The results are collected and can be visualized in a dashboard.
5. **Feedback collection:** Some data generators can use as input the results of previous test runs.

One of the experiments carried out to validate RESTest was to run it against APIs of the following popular services: GitHub, Foursquare, Marvel, Stripe, Tumblr, Yelp and YouTube (MARTIN-LOPEZ; SEGURA; RUIZ-CORTÉS, 2021). The authors of the research generated more than 90K test cases. Of those, 30% failed (uncovering errors in all APIs being tested). Many different types of issues and errors were found: server errors, client errors (in response to valid inputs) and mismatches between the API documentation and its actual implementation.

As future work, the authors intend to extend RESTest in many different ways. One idea is to offer more test data generators and test case generation strategies. Another plan is to support other specifications besides OpenAPI, even allowing RESTest to also be able to generate and run non-functional test cases.

### 3.6 *Competing OpenAPI repositories and collections*

There does exist directories of APIs (e.g., RapidAPI<sup>17</sup>) and even collections of OpenAPI specifications (e.g., APIs Guru<sup>18</sup>). However, these existing solutions are not a good fit for researchers for two important reasons.

---

<sup>17</sup> <https://rapidapi.com>

<sup>18</sup> <http://apis.guru>

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 kinds of repositories is simply on creating OpenAPI specifications for existing web services, without a concentrated effort (such as in the case of OAS DB, presented in Chapter 5) 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).

### 3.7 *Final remarks*

The literature review highlights some interesting facts. First, it appears to be a tendency of creating novel models and specifications instead of using the ones that are popular in industry. Iversen (2018) pinpoints the rationale behind this trend: by doing so, researchers are able to focus on that which is most relevant to their research and do not have to deal with the complexities of real world specifications (such as OpenAPI).

Regarding the issues commonly found in REST APIs, one can conclude that they are not yet crystal clear. As shown by the results of this literature review, so far there are not that many studies focusing on finding bugs and violations of REST API best practices. The need of further investigating issues that plague REST APIs in general is indeed also directly pointed out by some of the reviewed works (e.g., see Atlidakis, Godefroid and Polishchuk (2019)).

To answer the third and final question formulated in Section 3.1, let us look at the datasets used in the reviewed studies. With the exception of one study (which is Ed-Douibi, Izquierdo and Cabot (2018)), all others use custom datasets created by their own research team (at least when dealing with REST APIs in the case of studies concerned with multiple architectures). The OpenAPI specifications repository APIs Guru (used in Ed-Douibi, Izquierdo and Cabot (2018)) is not highly suited for research usage. In Chapter 5, the reasons this repository is not a good fit are thoroughly explained. One can argue that this lack of a standard dataset is detrimental to advancements in the field for two main reasons: first, researchers spend valuable time building their own datasets;

and, besides that, the usage of different data makes the comparison between studies more challenging.

Regarding tools that aim to find issues in REST APIs, there are other relevant works besides RESTler (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019; GODEFROID; HUANG; POLISHCHUK, 2020; GODEFROID; LEHMANN; POLISHCHUK, 2020) and RESTest (MARTIN-LOPEZ; SEGURA; RUIZ-CORTÉS, 2021). Alonso *et al.* (2022) introduces ARTE, a realistic test input generator that can be integrated with API test generators. In the aforementioned work, ARTE is integrated with RESTest. ARTE is able to generate realistic test inputs – e.g., a valid ISO 3166 country code for a country code parameter in an API – by extracting semantic information from an API’s specification. In the case of OpenAPI specifications, this information is extracted from a parameter’s name and description. After doing so, ARTE is able to query semantic knowledge databases during the process of input generation. By following the approach just described, ARTE was able to detect confirmed bugs in the APIs of real-world services, such as Amadeus (a hotel booking service) and DHL (a logistics company).

Baniaş *et al.* (2021) introduces a tool capable of generating functional tests for a REST API having as input its OpenAPI specification. The tool is also able to evaluate the performance of the API by measuring the response times for each API endpoint. Test cases can be generated using a set of different modes, ranging from totally automated to requiring the user to give sample values for parameters without available examples in the API’s specification. The approach described in Baniaş *et al.* (2021) is validated by two case studies: the first uses an API dedicated to American football data with 42 paths and 38 different parameters; the second, uses a collection of 30 different APIs extracted from APIs Guru.

Corradini *et al.* (2022) presents RestTestGen, a new approach for automatically generating tests for REST APIs from their corresponding OpenAPI specifications. RestTestGen is able to generate both nominal and error test cases. The former class of tests respects what is specified in the API’s specification, while the later purposefully creates tests that violate the expectations given in the OpenAPI specification as a way to try to force the API into error states. The tool also introduces the concept of operation dependency graphs, which is a method for detecting dependencies between different parameters and only running certain test cases when all mandatory dependencies are met. RestTestGen

was validated against 116 real-world APIs and was able to uncover many different defects in them.

Mirabella *et al.* (2021) introduces an approach for predicting the validity of generated API test inputs using Artificial Intelligence techniques. It is very common for API endpoints to include parameters that are interdependent (i.e., some combinations of values for them produce invalid requests). Google Maps API, for instance, requires that requests that inform a value for the `location` parameter also include the `radius` parameter as well (MIRABELLA *et al.*, 2021). As a consequence of these interdependencies, it becomes more challenging to randomly generate valid test inputs. Mirabella *et al.* (2021) proposes to train a deep learning model on previous API request / response pairs. This approach results in a high level of accuracy when predicting if a generated input is valid (ranging from 86% to 100% accuracy in some well known real-world REST APIs).

Karlsson, Causevic and Sundmark (2020) describes a proof of concept tool — QuickREST — that uses a two-folded strategy for generating tests for REST APIs: it both creates random tests and tests that conform to the corresponding OpenAPI specification. By going beyond the verification of HTTP response codes and also checking a response’s properties, QuickREST was able to detect issues in real world APIs. As its validation, QuickREST is able to reproduce the same bugs found by another tool — RESTler (ATLIDAKIS; GODEFROID; POLISHCHUK, 2019) — in the popular open source project GitLab<sup>19</sup>.

There are some other initiatives that, although tangential to our specific area of research and purpose, are worth mentioning. Kistowski *et al.* (2018) introduces TeaStore, a reference micro-services application that can be used to benchmark micro-service deployment strategies and also as a model to researchers and industry practitioners working in this field. Another example that serves a similar purpose is the Sock Shop demonstration application<sup>20</sup>. It is important to stress that a) our research is specifically about REST APIs and the OpenAPI specification, not micro-services and b) that both TeaStore and Sock Shop were created with the purpose of being best practices reference models, not with the aim of supporting researchers working with the detection of issues and anti-patterns (which is precisely the purpose of our work).

---

<sup>19</sup> <https://gitlab.com/>

<sup>20</sup> Sock Shop Demo

Efforts to build shared experimentation infrastructure seem to have yielded positive effects in other research fields. Hyunsook Do et. al. (DO; ELBAUM; ROTHERMEL, 2005) demonstrated 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). Another work 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 (JUST; JALALI; ERNST, 2014).

## 4 REST anti-patterns

As discussed in the literature review in Chapter 3, there are already some studies that catalog REST API anti-patterns (see in particular Petrillo *et al.* (2016)). We defend, however, that there are relevant anti-patterns present in real world APIs that are not yet documented in the literature.

In this chapter, we present a novel REST anti-pattern. We also explore in detail another REST anti-pattern that is already cataloged in the literature. By studying these two anti-patterns, we intend to give a good sense of the kinds of anti-patterns that will be present in the specifications that are part of OAS DB, presented in Chapter 5. Both anti-patterns explained in this chapter are already present in OAS DB.

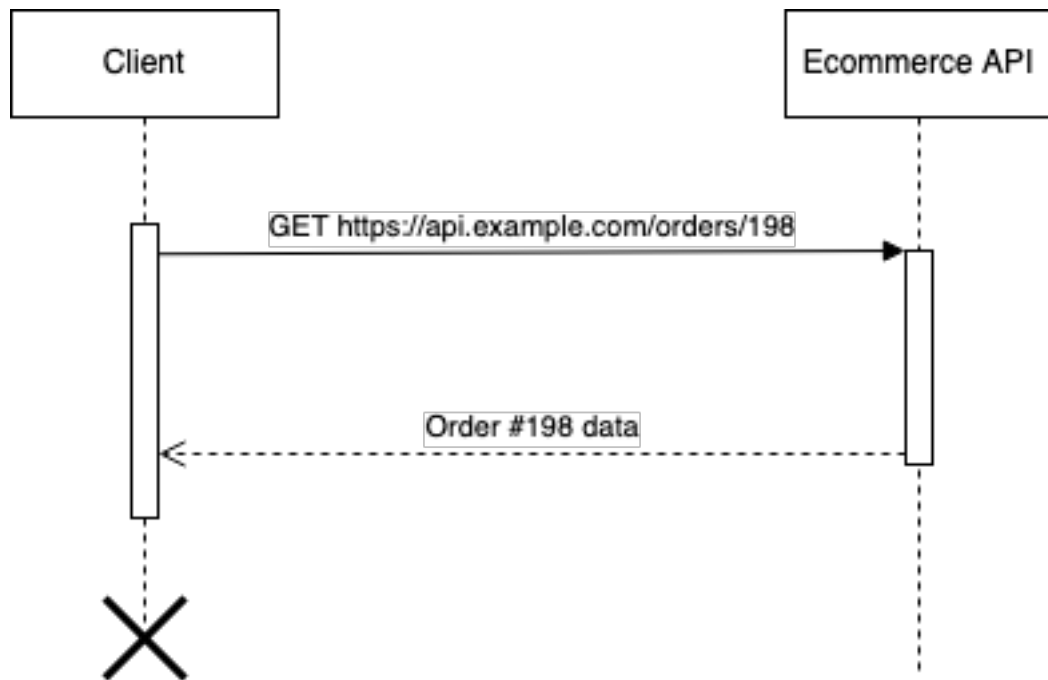
### 4.1 *Proposed new anti-patterns*

#### 4.1.1 Sequential integers as resource ID

It is common knowledge that most resources in a software system are distinguishable by some kind of unique identifier given to them. For example, an order in an ecommerce system may have the integer *198* as its ID. Using integers as IDs is a common practice because of ease of implementation and human comprehensibility.

For a deeper understanding of the issue, let us continue with this example and imagine that this same ecommerce system exposes a REST API. Let us suppose that there is one endpoint that allows one to retrieve information about an order by providing its integer ID.

Figure 8 – A client requesting an order's data by providing its integer ID



Source: Alex Braha Stoll, 2022

Sequential integer IDs are a problem because they are easily fabricated and they leak information about system internals. Let us explore in more details each one of these characteristics to understand how they are a bad practice both from a security standpoint and from a business perspective.

Sequential integers IDs are detrimental from a security standpoint

If one can access information about resources by providing its ID — as we are seeing in the ecommerce API example — this means that having IDs that are easily forged (such as in the case of sequential integers) gives an attacker an obvious channel to try to explore. Considering our example endpoint for retrieving orders, an attacker has two paths to try to explore:

1. If the attacker is not authorized to have any access at all to the API, the attacker can generate integer IDs (which are probably valid and correspond to real orders) hoping to get data from orders due to a potential error in the implementation of the authentication and authorization mechanism of the API;
2. If the attacker has authorization to access at least part of the orders, the attacker can generate integer IDs to try to get a hold on data that it should not have access

to. As an example, imagine a scenario in which a vendor can only access data from orders that include items s/he sold, but tries to generate IDs to access data from orders in which s/he is not participating in.

Both of these exploits rely on errors in the authentication / authorization mechanism of the system under attack. However, those types of errors are not unheard of. There are CVEs reporting cases in which the channel just described was successfully used to compromise real world software (CVE-2015-8542, 2015).

Sequential integers IDs are detrimental from a business perspective

From a business perspective, we defend that the usage of sequential integer IDs is also detrimental. Let us continue considering our fictional ecommerce system and its API. Since the orders are identified by sequential integers, this means that we are leaking information that could be used by a competitor to estimate the sales volume of the company running the ecommerce. There are probably multiple ways of doing so, but here are two feasible strategies:

1. The competitor makes a purchase from the ecommerce. A week later it makes a new purchase. The delta between order numbers can be used as a rough estimate of the weekly sales volume;
2. If the competitor knows a cooperating vendor that has access to the API or for some reason has obtained access to the API itself, the fact that the orders are identified by sequential integers may probably also allow them to devise a strategy to estimate the sales volume of the ecommerce.

Scenarios in which it is appropriate to use sequential integers

It is important to note that there are scenarios in which the usage of sequential integer identifiers is perfectly valid. It is the responsibility of the designers and of the implementers of a given system to judge if using sequential integers in a specific context may cause the type of security and business disadvantages discussed previously. Although not an extensive list, here are some contexts in which using integers is appropriate:



1. Internal systems used only by trusted parties;
2. Contexts in which performance requirements are extreme and the cost of generating sequential integers has been verified to be considerably lower than that of using UUIDs;
3. Systems that run in an environment in which no reliable UUID generator implementation is available.

## 4.2 Discussing an example of a cataloged REST anti-pattern

### 4.2.1 Sensitive information in the path or in the query string

Not all APIs — or all resources accessible through an API — are to be public. When considering different mechanisms for authentication and authorization of an API user, one possible strategy is to require a token to be sent together with the rest of the data necessary to make a request to a given endpoint.

To illustrate this concept, let us consider again a fictitious ecommerce API. However, this time let us imagine an endpoint that allows a vendor to access data on a given customer, as long as the customer has bought at least one item from that vendor in the past. In order for a vendor to access information on a customer, it needs to provide a customer token, which is only obtainable if the requirement explained before is met. Such an endpoint could be described by the following segment of OpenAPI specification:

Figure 9 – Sensitive information included in the path or in the query string, a cataloged REST anti-pattern

```

1  /customers/{customer_token}:
2  get:
3    summary: Allows a...
4    tags:
5      - customer
6    parameters:
7      - name: customer_token
8        in: path
9        required: true
10       description: A token...
11       schema:
12         type: string

```

Source: Alex Braha Stoll, 2022

In the segment above, we see that the token is required as part of the path of the endpoint. Another common strategy is to require some kind of token or secret as part of the query string. We can find multiple examples of this kind of approach in specifications found at the APIs Guru repository<sup>1</sup>.

The issue with this approach — as explained in Iversen (2018) — is that the URI (which the path and the query string are parts of) is not encrypted as the request data travels through the Internet, even when using a secure protocol like HTTPS.

How can this anti-pattern be avoided? In other words, how could one send a required authentication token without taking the risk of exposing it? One possible solution is to send the token as part of the header of the HTTP request, which is encrypted and thus secure when a protocol like HTTPS is employed.

### 4.3 *Final remarks*

In this chapter, we did a deep dive on concrete types of REST anti-patterns. In section 4.1 we presented a novel anti-pattern not yet documented in the literature. In section 4.2.1, we then discussed with details one example of anti-pattern that is already catalogued.

Now that we are familiar with REST anti-patterns, we are ready to start exploring how to inject them into OpenAPI specifications and also mock implementation APIs. In the next chapter, we will present OAS DB, the main product of our research. OAS DB is capable of generating complete OpenAPI specifications and their corresponding mock API implementations. During this synthetization process, OAS DB can inject into these assets multiples types of REST anti-patterns and other kinds of issues and faults.

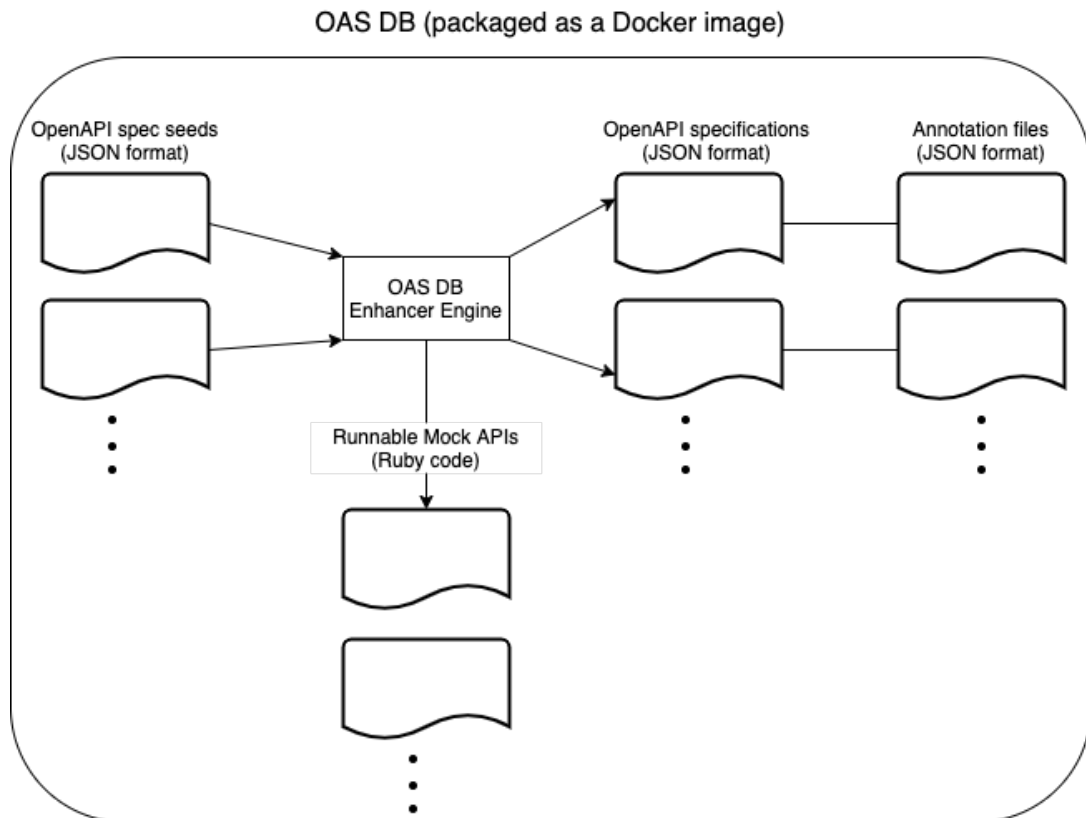
---

<sup>1</sup> <http://apis.guru>

## 5 OAS DB: A generator of annotated specifications and mock APIs to support OpenAPI research

OAS DB stands for OpenAPI Specifications Database. It is a tool capable of generating a collection of synthetic OpenAPI specifications and their mock API implementations, together with annotation files describing both. OAS DB is openly available<sup>1</sup> at GitHub. The specifications, mock API implementations and annotation files are automatically created from segments of OpenAPI specifications. These segments — which we call OpenAPI specification seeds — are also one of the contributions of the present research.

Figure 10 – Overview of OAS DB



Source: Alex Braha Stoll, 2022

As illustrated in Figure 10, the OAS DB Enhancer Engine is the heart of OAS DB. As the name suggests, this is the component responsible for transforming an OpenAPI specification seed into the three final products offered by OAS DB: a complete OpenAPI specification, an executable mock API implementation of the specification and finally an annotation file describing which anti-patterns appear in the specification (if any) and which issues and faults in the mock API (if any).

<sup>1</sup> <https://github.com/alexbrahastoll/oas-db>

The remainder of this chapter is organized in the following fashion: section 5.1 contains a quick refresher on the OpenAPI specification and also presents the concept of OpenAPI specification seeds, the initial input needed by OAS DB. Section 5.2 discusses the generation of specifications from seeds. Section 5.3 explores the generation of mock API implementations. Section 5.4 discusses annotations files, responsible for indicating which issues are to be found in corresponding specifications and mock API implementations. Section 5.5 explains in details the information flow through the different components of the tool and how one can use it to generate specifications, mock API implementations and annotation files. Section 5.6 briefly instructs one on how to contribute to the development of OAS DB. Finally, section 5.7 compares OAS DB with similar existing solutions.

### 5.1 *OpenAPI specification seed*

To recapitulate, OpenAPI<sup>2</sup> is a format devised for describing all the resources that can be accessed and manipulated through a REST Web API. An OpenAPI specification seed is based on this format and is the raw material used by OAS DB to generate full-blown OpenAPI specifications and their corresponding annotation files and executable mock API implementations. This file type is not part of the OpenAPI standard and is a novel idea that is part of the research herein presented.

When designing the OpenAPI specification seed file format, we aimed to identify the parts of OpenAPI essential for generating a complete specification, together with a mock API implementation. As a consequence, the format that we devised contains some of the components that are part of a complete OpenAPI specification. In other words, an OpenAPI specification seed is a subset of OpenAPI version 3.0.3. Although an OpenAPI file can be represented in JSON (JavaScript Object Notation) or YAML (YAML Ain't Markup Language), a specification seed must be a JSON file.

Currently, a seed must only contain a single entity and it considers all its fields to be mandatory for record creation. In the case of updating a record, it is assumed that at least one of the expected fields must be present. Besides that, all the properties of the main entity must be of one of the following types: integer, number, string and boolean. In the future, we may expand what the format allows in order to be able to generate more complex complete specifications and mock APIs. Figure 11 is an example of the current

---

<sup>2</sup> <https://www.openapis.org/>

version of this file format. It shows a seed specifying a fictitious API to report and manage incidents that happened in an online service (e.g. application downtime).

A valid seed must contain the following components:

1. **the info object:** it contains basic information about the API, such as its title, description and version (OpenAPI..., 2020).
2. **the schema object:** it has the fields and data types of the main entity of the API.
3. **the example key:** it is one of the keys of the the **schema** object. It must hold a valid instance of the entity being represented.

Figure 11 – An OpenAPI specification seed that describes the kernel of an incident reporting API.

```

1  {
2    "info": {
3      "title": "Incident response API",
4      "description": "API that allows one to report and manage incidents that happen in an online service (e.g. an ecommerce).",
5      "version": "0.1"
6    },
7    "components": {
8      "schemas": {
9        "Incident": {
10          "type": "object",
11          "description": "An incident.",
12          "properties": {
13            "title": {
14              "description": "A title to identify the incident.",
15              "type": "string"
16            },
17            "service_id": {
18              "description": "The ID of the affected service.",
19              "type": "integer"
20            },
21            "assignee_id": {
22              "description": "The ID of the colaborator assigned to deal with the issue.",
23              "type": "integer"
24            }
25          },
26          "example": {
27            "title": "30 minutes outage due to someone tripping on the server's power cable.",
28            "service_id": 101,
29            "assignee_id": 11
30          }
31        }
32      }
33    }
34  }
```

Source: Alex Braha Stoll, 2022

## 5.2 OpenAPI specifications

OAS DB is capable of generating samples of OpenAPI specifications containing known anti-patterns. Let us illustrate this feature with a short example. In REST APIs, it

is a good practice not to allow sensitive information in the query string of a given endpoint (IVERSEN, 2018). 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. In Figure 12, we observe part of a generated OpenAPI sample that contains the anti-pattern just described.

Figure 12 – Sensitive information (the customer token) included in the query string, one of the new anti-patterns we propose.

```

1  /customers/{customer_token}:
2  get:
3    summary: Allows a...
4    tags:
5      - customer
6    parameters:
7      - name: customer_token
8        in: path
9        required: true
10       description: A token...
11       schema:
12         type: string

```

Source: Alex Braha Stoll, 2022

### 5.2.1 REST anti-patterns

#### Selection of REST anti-patterns

One of the main goals of OAS DB is to include a diverse and realistic set of anti-patterns. Therefore, when implementing the components responsible for generating specifications including anti-patterns, we selected anti-patterns that are shown to be more present in real APIs (see Petrillo *et al.* (2016); Ed-Douibi, Izquierdo and Cabot (2018); Brabra *et al.* (2019); Palma, Moha and Guéhéneuc (2018); and Atlidakis, Godefroid and Polishchuk (2019)). Here is the list of anti-patterns OAS DB is currently capable of injecting into an OpenAPI specification:

1. **Crudy URI:** Injects a HTTP action verb (or a synonym) into the resource's path (e.g., `get_payments`);
2. **Amorphous URI:** Adds superfluous characters to the resource's path (e.g., a file type suffix such as `.xml`);

3. **Ignoring status code:** Responses use inappropriate HTTP codes (e.g., an endpoint returning 204 instead of 201 after a resource is created);
4. **Inappropriate HTTP method:** Requests expect inappropriate HTTP methods (e.g., an endpoint for reading a resource expecting POST instead of GET);
5. **Invalid examples:** Generates request examples that do not comply to the related objects' schema;
6. **Sensitive info in the path or query string:** Sensitive info (e.g., a token) is included in paths or query strings.

## Categories

Each selected anti-pattern / issue was categorized as negatively affecting one or more attributes of software quality, be it product quality attributes, be it quality in use attributes, as defined by ISO 25010:2011. The list of anti-patterns / issues and the corresponding affected attributes is shown in table 3.

Product quality attributes are the following: functional suitability, performance, compatibility, usability, reliability, security and maintainability. Quality in use attributes are: effectiveness, efficiency, satisfaction, freedom from risk and context coverage (ISO 25010, 2011).

Table 3 – Anti-patterns and the ISO 25010:2011 attributes affected

Name	Negatively affects
Crudy URI	Compatibility, efficiency
Amorphous URI	Compatibility
Ignoring status code	Compatibility, effectiveness, efficiency
Inappropriate HTTP method	Compatibility, performance
Invalid examples	Maintainability, efficiency
Sensitive info in the pqs	Security
Invalid payload	Reliability
Unexpected payload root node	Reliability
Payload missing keys	Reliability
Payload extra keys	Reliability
Payload wrong data types	Reliability
Broken record deletion	Functional suitability, effectiveness

Source: Alex Braha Stoll, 2022

### 5.3 Mock API implementations

#### 5.3.1 Code generation

One of the products of processing an OpenAPI specification seed is a mock API implementation. This is a self-contained file that, when executed, spins up a server that is able to receive requests and generate responses, according to what the corresponding OpenAPI specification file dictates. The generated API will have the four common record manipulating operations: create, read, update and destroy.

Figure 13 – A segment of a mock API implementation. This excerpt shows a GET endpoint that is able to retrieve a record when given its ID.

```

1  get '/incidents/:incident_id' do
2    request.body.rewind
3
4    obj_id = Integer(params['incident_id'])
5    found, obj = api_helper.read_obj(obj_id)
6
7    halt 404 unless found
8
9    res_body = JSON.dump(obj)
10   res_header = { 'Content-Type' => 'application/json' }
11
12   [200, res_header, res_body]
13
14 rescue ArgumentError
15   halt 404
16 end

```

Source: Alex Braha Stoll, 2022

Figure 13 shows a fragment of a generated API implementation. At lines 4 — 7, we try to retrieve the requested object by using the ID supplied as a parameter. If the ID given does not correspond to an existing object, we halt the processing and respond with the HTTP error code 404 (meaning record not found). Lines 9 — 12 handle the scenario in which the supplied ID does correspond to an existing object. When that is the case, we build the head and body of the response and return it with a 200 HTTP status code (meaning success).

The executable API is a Ruby source file mocking what is specified in the corresponding OpenAPI specification: the API endpoints, success and error HTTP codes, expected payloads and so on. Although the mock implements what is specified in the associated OpenAPI specification, it is generated directly from the OpenAPI seed (as is the case for both the generated specification and annotation files). As a consequence, we have the power and flexibility of purposefully injecting bugs and other issues in the



generated code. This is valuable when testing a tool, since we are able to run it against code that has known issues.

Having these mock API implementations makes OAS DB a complete solution because they allow researchers to test tools that use dynamic analysis techniques (i.e., techniques that interact with a running system). One such tool is RESTler, presented in Subsection 3.5.1.

The mock API implementation code leverages Sinatra<sup>3</sup>, an open-source web framework for the Ruby programming language. By using the framework, it is possible not only to create single-file, self-contained web systems and web APIs, but also much larger systems. The framework is very straightforward to use and because of that it lends itself well for some special cases, like for example automatically generating the code for a web system or API. Section 2.8 discusses the framework with more details.

These self-contained mock implementations also include two features that may help a researcher while using them to evaluate tools that need to interact with a running API: a simple in-memory database and a field validation mechanism. Subsection 5.3.3 discusses the former, while subsection 5.3.4 presents the latter.

### 5.3.2 Fault and issue injection

When generating mock API implementations using OAS DB, one has the ability to have bugs and issues injected in the API code. These can be specified by providing the generator with certain parameters, listed in detail in OAS DB's documentation.

Figure 14 shows a snippet of API code generated including one of the issues available to be injected. At line 1, we can see that this generated API include the `payload_missing_keys` issue. As a consequence, the API will crash and return an HTTP error code 500 every time a payload with missing required keys is sent to the server. Note that an API generated without this injected issue will be able to gracefully handle this error scenario. At lines 5 — 18, we have the method responsible for payload sanitization. Note that in line 6, we check to see if the API was generated with the aforementioned issue. When the API is free of this fault, the remaining of this method is responsible for deleting keys that are not present in the object's schema, defined in the corresponding OpenAPI specification. Finally, lines 22 — 26 show the method responsible for crashing

---

<sup>3</sup> <https://www.sinatrarb.com>

Figure 14 – A segment of a mock API implementation. This excerpt shows a method whose implementation includes injected issues.

```

1  API_ISSUES = ['payload_missing_keys'].freeze
2
3  # ...
4
5  def sanitize_payload(payload, schema)
6    injector.payload_missing_keys_err(payload, schema)
7    injector.payload_extra_keys_err(payload, schema)
8
9    sanitized_payload = payload.deep_dup
10
11    payload.each do |name, value|
12      sanitized_payload.delete(name) unless schema[name].present?
13    end
14
15    injector.payload_missing_keys_err(sanitized_payload, schema)
16
17    sanitized_payload
18  end
19
20  # ...
21
22  def payload_missing_keys_err(payload, schema)
23    return unless API_ISSUES.include?('payload_missing_keys')
24
25    raise PayloadMissingKeysError if payload.keys.length < schema.keys.length
26  end

```

Source: Alex Braha Stoll, 2022

APIs that include the `payload\textunderscore missing\textunderscore keys` issue when payloads without all required keys are received. Note that this is the method that is called at the previously explained line 6.

In general, an injected issue or bug forces a failure in a situation that is normally gracefully handled (e.g., when the API receives a payload containing extra keys). OAS DB currently allows the following issues and bugs to be injected into a generated API implementation:

1. **invalid\_payload:** Forces the API to raise an unhandled exception (and respond with HTTP code 500, internal server error(RFC 7231, 2014)) when an invalid JSON payload is sent to the server while creating or updating a record.
2. **unexpected\_payload\_root\_node:** The API responds with HTTP code 500 when a JSON payload whose root is not an `object` is sent to the server while attempting to create or update a record.

3. **payload\_missing\_keys:** Code 500 when a JSON payload does not include all the required keys for creating a record.
4. **payload\_extra\_keys:** The API returns 500 when a payload has extra, unexpected keys. Applicable to the create and update operations.
5. **payload\_wrong\_data\_types:** The API responds with 500 when a payload is a valid JSON and has the required keys, but has at least one key whose data type is incorrect (e.g. an integer where a string is expected).
6. **broken\_record\_deletion:** The API responds with success (HTTP code 200) when asked to delete a record, but the record is not actually destroyed.
7. **invalid\_examples:** Different from the other available issues, this is not injected in the mock API implementation, but rather in its corresponding OpenAPI specification. Invalid examples are included in the specification (e.g. the example for the create operation has a payload missing required keys).

### 5.3.3 In-memory database

Every generated API includes a simple embedded key-value in-memory database implementation, which allows them to keep state between different HTTP requests. This feature is essential for detecting issues that are only manifested after a sequence of interactions with the server.

As an example, let us consider a tool that is trying to detect whether there are cases in which it is possible to use a resource after its destruction. One way of finding such an issue would be to first create a record, then delete it and finally try to read it. Without a database, we would not be able to persist the record in the first place, so the generated API would be useless when employed in evaluating a tool trying to detect the issue aforementioned.

Figure 15 shows the bulk of the code that implements the embedded key-value store. At line 1, we have `create_obj`, the method responsible for creating a new object. It leverages the `next_id` method, which keeps track of generated IDs and returns a new and unique one every time it is called. Then, at the end of `create_obj`, we store the new object in the instance variable `ds`, which holds a Ruby Hash (i.e., a dictionary data structure). At lines 8 and 13, we have `read_obj` and `update_obj`, respectively responsible for reading and

updating a previously stored object from the dictionary held at the instance variable `ds`. Both these methods implement mechanisms to check whether the object one is trying to manipulate actually exists. Finally, at line 21 we have the `delete_obj` method, responsible for deleting an object. Note, at line 23, that this method is capable of purposefully failing to remove the object when APIs include the `broken_record_deletion_err` issue (refer to Subsection 5.3.2 for explanation on injectable faults and issues).

Figure 15 – The bulk of the code implementing the embedded key-value store. The implementation leverages Ruby’s Hash data structure.

```

1  def create_obj(payload)
2    id = next_id
3    obj = payload.merge({ 'id' => id })
4    ds[id] = obj
5    [true, obj]
6  end
7
8  def read_obj(key)
9    obj = ds[key]
10   [!obj.nil?, obj]
11 end
12
13 def update_obj(key, payload)
14   obj = ds[key]
15   return false if obj.nil?
16
17   ds[key] = ds[key].merge(payload)
18   true
19 end
20
21 def delete_obj(key)
22   return false if ds[key].nil?
23   return true if injector.broken_record_deletion_err
24
25   ds.delete(key)
26   true
27 end

```

Source: Alex Braha Stoll, 2022

#### 5.3.4 Field validation

The generated mock API implementations also include automatic field data type validation. This is a mechanism that validates — when one is trying to create or update a record — if the payload provided includes all the required fields, each having data of the

expected type (e.g., a field specified as integer cannot be passed to the server holding a string).

This feature is relevant for tools that attempt to verify if an API is actually honoring its specification. An API may have a specification that defines the data types expected for different fields, however due to a fault in implementation (or due to a wrong specification in the first place) it may be possible to create records without honoring the expected data types or not following an expected data type may cause the API to altogether crash.

OAS DB's field data type validation is implemented by leveraging Ruby's standard data conversion methods. We generate code that maps OpenAPI data types to these Ruby conversion methods, allowing us to check each field according to its declared data type.

Figure 16 – Part of the code implementing the data type validation mechanism.

```

1  OAS_RUBY_DATA_VALIDATION = {
2    'integer' => ->(data) { Integer(data) },
3    'string'  => ->(data) { raise ArgumentError if String(data).length == 0 },
4    'number'  => ->(data) { Float(data) },
5    'boolean' => ->(data) { raise ArgumentError unless [true, false].include?(data) }
6  }.freeze
7
8  # ...
9
10 def validate_field(name, value, schema)
11   OAS_RUBY_DATA_VALIDATION[schema.dig(name, 'type')].call(value)
12   true
13 rescue StandardError
14   false
15 end

```

Source: Alex Braha Stoll, 2022

Figure 16 shows part of the code responsible for the mechanism herein explained. At line 1, we assign to the `OAS_RUBY_DATA_VALIDATION` constant a dictionary to help with data type validation. The dictionary keys are OpenAPI data types (e.g., `string`), while the values are Ruby lambda functions. These are similar to what we call anonymous functions in other programming languages. These constructs allow us to succinctly define functions to convert string values sent by the client into the expected data type. At line 10, we define the `validate_field` method, responsible for validating each field passed by the client when attempting an operation that requires a payload (e.g., creating a record). This method internally uses the functions we have just discussed. At line 11, we see that in order to know which data type conversion function to call, we refer to the corresponding

OpenAPI specification. There, we can find the expected fields for an operation requiring a payload and also the data type of each field.

#### 5.4 Annotation files

Each OpenAPI specification present in OAS DB is accompanied by an annotation file in the JSON format. Each annotation file describes all anti-patterns found in the associated specification, including the segment and line in the OpenAPI file responsible for each violation. Figure 17 shows an annotation file documenting the anti-patterns present in the associated OpenAPI sample.

These annotations provide a way for researchers to automatically verify the effectiveness of new techniques and tools. To make this point clearer, let us suppose a scenario in which a researcher is using a repository without annotations to verify a tool that aims to detect anti-patterns. Since it is not known *a priori* which anti-patterns are actually present in this repository of samples, the researcher will have to analyze every sample manually in order to determine the true and false positive and negative detections of the tool being evaluated.

When using OAS DB in this same scenario, that effort will not be necessary because each sample is accompanied by an annotation file that lists all the anti-patterns to be found. Moreover, since the annotation files follow a defined structure, it is straightforward to create a script to automatically compare the results obtained by the tool under evaluation and the anti-patterns that are actually present in each specification. Such a script could naturally be used repeated times (for the same or related tools), potentially saving a lot of labor for the researcher / research team.

#### 5.5 Using OAS DB: from seed to running mock API

To facilitate the usage of OAS DB, we distribute it packaged as a Docker container (see Section 2.9 for a brief introduction on Docker). This strategy will allow researchers to run it on their local machines or have it deployed and running on their cloud environment of choice. The Docker container will contain not only OAS DB itself, but also all of its dependencies. As a consequence, the process of generating and running mock APIs will

Figure 17 – An annotation file documenting that the associated OpenAPI specification has one anti-pattern

```

1  {
2    "version": "oas-db-0.1",
3    "annotationTarget": "ecommerce.yml",
4    "violations": [
5      {
6        "type": "sensitive_info_pqs",
7        "categories": ["security"],
8        "offender": "paths./customers/{customer_token}",
9        "location": 1
10     }
11   ]
12 }

```

Source: Alex Braha Stoll, 2022

be facilitated: researchers will only need to be able to run the provided container either locally or in their preferred cloud environment (there is extensive documentation on the Docker website<sup>4</sup> on how to run and interact with applications packaged as containers).

OAS DB will also be accompanied by documentation detailing on how to use it. This documentation will be made available in the same GitHub repository<sup>5</sup> hosting the project. In summary, researchers and practitioners interested in using OAS DB in their projects should probably advance in the following order: 1) to become familiarized with the versions of OpenAPI supported by OAS DB; 2) to have a general idea of the features of the tool; and 3) to get acquainted with the format of annotation files, since with the metadata present in them it is even possible to create solutions for automatically determining the accuracy of a tool under evaluation.

Now that we have gone through all components of OAS DB, we show an example of how one could use the tool, discussing along the way how information flows between the different components of OAS DB illustrated in figure 10.

### 5.5.1 CLI and configuration file

The first step is to have in hands a specification seed and a configuration file. In this walk through, let us use as an example the seed contained in figure 11, in section 5.1. The configuration file is a JSON specifying information such as which faults and issues are to be

<sup>4</sup> <https://docs.docker.com/>

<sup>5</sup> <https://github.com/alexbrahastoll/oas-db>

injected in the generated assets. Figure 18 shows a configuration file instructing OAS DB's Enhancer Engine to inject certain issues in the specification (the key `spec_issues` in the configuration) and mock API (key `api_issues`) that will be created. The issues the tool supports are the same listed and discussed in subsection 5.3.2. For detailed information on the configuration options available, please refer to OAS DB's documentation.

Figure 18 – An example of a configuration file expected by OAS DB's CLI.

```

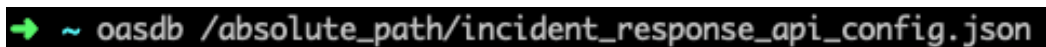
1  {
2    "oas_seed_abs_path": "/absolute_path/incident_response.json",
3    "mock_api_server_url": "http://localhost:3000",
4    "spec_issues": [
5      "invalid_examples"
6    ],
7    "api_issues": [
8      "broken_record_deletion"
9    ],
10   "generated_files_basename": "incident_response_invalid_examples"
11  }

```

Source: Alex Braha Stoll, 2022

The CLI included in OAS DB then reads these configuration options and passes them to the OAS DB Enhancer Engine, as shown in figure 10. Using the CLI is not mandatory, but it is probably the most straightforward way of using the OAS DB Enhancer Engine. If one is working in a Ruby project, however, one could directly import and use the OAS DB Enhancer Engine. Figure 19 shows an example of how one would call the OAS DB CLI passing the mandatory argument, which is the absolute path to a configuration file.

Figure 19 – An example of calling OAS DB's CLI from a terminal.



```

➔ ~ oasdb /absolute_path/incident_response_api_config.json

```

Source: Alex Braha Stoll, 2022

### 5.5.2 OAS DB Enhancer Engine

The OAS DB Enhancer Engine receives a specification seed and configuration options. As explained, part of what is given in the configurations is the set of faults and issues to be injected. As shown above, the issues that affect the OpenAPI specification to be generated are grouped as `spec_issues`. The ones that impact the mock API are specified as `api_issues`.



The engine has different subcomponents responsible for generating the OpenAPI section representing each one of the main operations one can perform on an entity: create, read, update and delete. These subcomponents, by default, will generate a specification conforming to REST and OpenAPI best practices.

The subcomponent responsible for generating the segment that describes the endpoint for the create operation, for example, will include an example of a valid payload, illustrating how one could create a record in practice (as the reader may remember, this example present in the generated OpenAPI specification comes from the seed). However, the generation can be affected depending on the issues to be injected. Figure 18 — which we are using as an example for this section — includes the issue `invalid_examples`. In this case, then, the aforementioned subcomponent would generate an *invalid* example when generating the create operation in the OpenAPI complete specification. Figure 20 shows the function responsible for generating an invalid example if the `invalid_examples` was specified as one of the issues to be present in the generated OpenAPI file.

Figure 20 – Function that generates an invalid example by removing keys from the valid example provided in the OpenAPI seed.

```

1  def gen_example(sample, breadcrumb)
2    return sample.base_resource_example unless raffled_antipatterns.include?('invalid_examples')
3
4    sample.annotation.add_antipattern('invalid_examples', breadcrumb)
5    sample.base_resource_example.deep_dup.slice(sample.base_resource_example.keys.first)
6  end

```

Source: Alex Braha Stoll, 2022

For the generation of the mock API implementation — a Ruby's Sinatra self-contained API, as discussed in section 5.3 — a similar architecture is followed. As is the case in the creation of the OpenAPI complete specification, the code responsible for generating each endpoint (again, for the create, read, update and delete basic operations) checks whether issues and faults are to be injected or not. Figure 18, our example configuration being used in this section, does include the `broken_record_deletion` fault. As explained in subsection 5.3.2, this issue causes the delete endpoint to fail at deleting the record but at the same time to respond as if the record was successfully deleted. Figure 21 shows that the function responsible for deleting a record does not actually destroy the object if the `broken_record_deletion` issue was injected in the generated mock API.

As the OpenAPI specification and mock API are being generated, the OAS DB Enhancer Engine keeps record of which issues were injected. This information will then

Figure 21 – Function that purposefully fails to delete a record if this is one of the faults the user wished to inject in the mock API.

```

1  def delete_obj(key)
2    return false if ds[key].nil?
3    return true if injector.broken_record_deletion_err
4
5    ds.delete(key)
6    true
7  end

```

Source: Alex Braha Stoll, 2022

be used to generate the annotation file, describing all the issues and faults present in the specification and in the mock API.

### 5.5.3 File Generation

The OAS DB Enhancer Engine internally represents all the components being generated — the full specification, the mock API and the annotation file — as Ruby data structures. Finally, when the generation process ends, these data structures are rendered to its respective file formats and saved to the disk.

Both the specification and the annotation are rendered as JSON. The API is rendered as executable Ruby code. After this process finishes and the files are saved in the disk, we have assets that are ready for usage. For example, one could execute the generated API to run tests against it or one could feed the complete OpenAPI specification to another OpenAPI related tool.

## 5.6 *Contributing to OAS DB*

OAS DB aims to be an open-source effort. Therefore, researchers and practitioners can craft new OpenAPI seeds and suggest their addition to the repository. Furthermore, interested parties can also contribute by enhancing the generation capabilities of OAS DB. This process is open to everyone, however approval of the overseers of the repository is fundamental. This is basically a manual peer-review done by the overseers of OAS DB to check not only technical details but also if the new suggested contributions satisfy quality requirements.

Besides this safeguard, deciding which new seeds to be created and added to OAS DB is of utmost importance. The seed should ideally be a realistic representative of a domain (e.g., ecommerce API) not yet present in OAS DB. If a seed belongs to a domain that is already represented in the repository, it may be more interesting to at least check if it is possible to enhance the existing seed somehow instead of creating a new one for the same domain. The same is true for new feature suggestions and collaborations: they must be aligned with OAS DB's overall objectives.

### 5.7 *Final remarks*

Currently there are a few other repositories of OpenAPI samples. One that appears in some studies (e.g., Ed-Douibi, Izquierdo and Cabot (2018)) is the APIs Guru repository<sup>6</sup>. This repository, however, has some features that in our view make it not a great fit for researchers.

The first one is the lack of annotations in the samples it contains, making it challenging for a researcher to check the performance of a tool tested against this repository (i.e., it is labor intensive to produce metrics because the samples are not annotated and therefore one has to manually analyze every one touched by the tool under evaluation).

The second characteristic of APIs Guru that in our estimation makes it not the best fit for researchers is the fact that the focus of the repository 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).

In this chapter, we presented in details OAS DB. As we have shown, this novel tool allows researchers and practitioners to generate annotated OpenAPI specifications and their corresponding mock implementations. OAS DB is open-source and can evolve over time with contributions from the technology community. In the next chapter, we will explore Oasis, a proof of concept tool that detects anti-patterns in OpenAPI specifications using static analysis techniques. As we explain in details in the next chapter, the main motivation for creating Oasis was to help validate OAS DB itself.

---

<sup>6</sup> <http://apis.guru>

## 6 Oasis: a tool for detection of anti-patterns in REST APIs

While reviewing the literature, we found tools that use static analysis techniques to detect issues on REST APIs (generally by analyzing an OpenAPI specification, but other methods are also used). For validating OAS DB’s capability of generating specifications with issues, our initial idea was to run an experiment with each one of these tools. By doing so, our intention was both to evaluate each tool and to show that OAS DB is indeed able to generate specifications containing the kind of anti-patterns and issues described in the relevant literature.

When trying to use the aforementioned tools, however, we were not able to conduct the experiments we planned for. UniDoSA (see subsection 3.3.3) has a web version (called WebRestpad<sup>1</sup>) that is online but is not working properly (as of January 2<sup>nd</sup>, 2022). While doing an initial exploration of the tool, we found that it was not able to fetch OpenAPI specifications from a given public accessible URI and consequently it was not able to even start any process of issue detection. We contacted the authors, who acknowledge the issue but did not present any plans for addressing it. Another work that looked promising was Brabra *et al.* (2019) (see subsection 3.3.2 for a summary of the article). While researching about the proof of concept tool the authors mention in their paper, we found a blog post discussing the work and offering a web version of the tool (SIMBAD Tool Blog Post, 2019). The link for the prototype of the aforementioned tool is broken and we did not get any response back from the main author responsible for the research. Finally, we tried to evaluate the tool introduced in Ed-Douibi, Izquierdo and Cabot (2018) (see subsection 3.4.1). This effort did not succeed as well. The tool has no documentation<sup>2</sup> and the author, in the comment section of a blog post about the research, comments that “the work on this project is currently stalled and the codebase is obsolete” (Modelling Languages Blog Post, 2018).

Having no other options, we decided to implement ourselves a tool for detecting some of the REST anti-patterns mentioned in the relevant literature. We did our best effort to implement detection techniques based on what appears on the different papers mentioned in chapter 3 (and to make reasonable inferences when details were not available). To this proof of concept tool, we gave the name Oasis (an allusion to **O**pen **A**PI **S**pecification).

---

<sup>1</sup> <http://webrestpad.sofa.uqam.ca/>

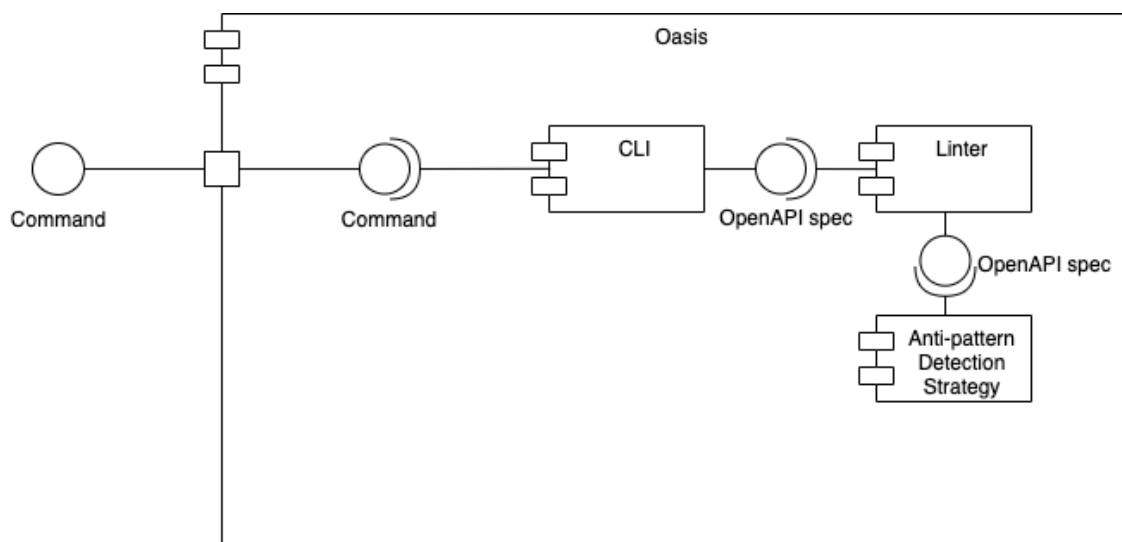
<sup>2</sup> <https://github.com/opendata-for-all/api-tester>

## 6.1 Overview of Oasis

Oasis is a proof of concept tool capable of detecting a few REST anti-patterns, according to implementation guidelines found in the relevant literature. Oasis is implemented using the Ruby programming language<sup>3</sup>. Ruby was chosen for two main reasons: firstly, because of the familiarity of the author of the present work with the technology, which speeds up the implementation of the tool; secondly, because its dynamic and interpreted nature allows for quick experimentation, which is ideal for a proof of concept project.

Oasis is a CLI tool. CLI stands for Command Line Interface, which means Oasis does not have a GUI (Graphical User Interface), but can only be operated via text commands. It is beneficial for projects like Oasis to expose a CLI because, unlike programs that only have a GUI, a command line interface allows for easier integration into existing software process pipelines. Consider, for example, a team that uses a script to run a series of different checks as part of their QA (Quality Assurance) process. Since Oasis is a CLI, it is convenient to modify the aforementioned script to run the checks made by Oasis as part of this QA process. Figure 22 shows a simplified and high-level overview of the components that compose Oasis.

Figure 22 – UML Component Diagram of Oasis



Source: Alex Braha Stoll, 2022

<sup>3</sup> <https://www.ruby-lang.org/en/>

1. **CLI:** This module is responsible for parsing text commands and calling the appropriate modules for the requested commands to be run (or for presenting error messages in the case of invalid commands).
2. **Lint:** Module responsible for having a list of all modules representing strategies for detecting anti-patterns. It is also responsible for calling the code which runs each anti-pattern detection strategy.
3. **Anti-pattern detection strategies:** A series of components each implementing an algorithm to check for the presence of a particular anti-pattern.

## 6.2 *Static analysis*

Oasis uses strategies that rely only on the analysis of the OpenAPI specification describing the API under analysis. The usage of static analysis is beneficial for multiple reasons. Firstly, it does not require one to have a deployed and running instance of the API under assessment. Secondly, it does not oblige the team running the API to provide canned responses for requests to the API (which is something fundamental when dynamic analysis techniques are in place). As a consequence of the two reasons stated above, running Oasis against an existing OpenAPI specification is a convenient way to check for REST API anti-patterns (at least those supported by Oasis and that are detectable by static analysis techniques).

## 6.3 *Anti-patterns currently detectable*

Our main purpose in creating Oasis was to validate OAS DB. Having this in mind, Oasis currently is able to detect only a few anti-patterns. While implementing the detection strategies, we made our best effort to stay as consistent as possible with the descriptions and guidelines offered in the literature.

### 6.3.1 Anti-pattern Amorphous URI

The Amorphous URI anti-pattern (from now on referred to as `amorphous_uri`) is described by Palma, Moha and Guéhéneuc (2018). It happens when an API's path include any characters besides lowercase alphabetic characters and the dash (—) character.

### 6.3.2 Anti-pattern Crudy URI

The Crudy URI anti-pattern (from now on referred to as `crudy_uri`) is present when an API's path includes one of the following verbs: create, read, update or delete. This anti-pattern is defined by Brabra *et al.* (2019).

### 6.3.3 Anti-pattern Sensitive Information in the Path or Query String

The Sensitive Information in the Path or Query String (hereinafter `sensitive_info_pqs`) is found on endpoints that include sensitive information (such as an authentication token) directly in the path or in the query string. This anti-pattern is discussed in detail at Iversen (2018).

## 6.4 *Comparison with other tools*

As previously explained, our main motivation in creating Oasis was the fact that similar tools found in the literature were simply broken or no longer being maintained. Nonetheless, let us make a quick comparison between these tools and Oasis.

Iversen (2018) introduces a tool that is able to detect issues and anti-patterns in REST APIs. However, the approach requires the API under analysis to be described using a custom description language introduced by the same study (instead of relying on a popular language such as OpenAPI or RAML). Besides that, Iversen (2018) mainly focus on security issues revolving around the usage of the JSON Web Token technology. By contrast, our study is not limited to security vulnerabilities, also including anti-patterns in other categories.

Ed-Douibi, Izquierdo and Cabot (2018) presents a tool that has as its only input requirement a valid OpenAPI specification. As stated in the study, however, the purpose of the tool is to generate test cases for the API under test, not to detect anti-patterns. There may be some overlap between these two different kinds of efforts, but in the end they are not aiming at exactly the same objective.

Brabra *et al.* (2019) not only concerns itself with REST anti-patterns, but also with the detection of violations of the OCCI standard. Adding to that, another considerable distinction when comparing that research with Oasis is the fact that their tool relies on models manually built after analysis of the documentation of each REST API. The tool introduced at Brabra *et al.* (2019) does not support OpenAPI specifications and requires this extra step of manually building a model from scratch for every new API that one wants to assess.

Palma, Moha and Guéhéneuc (2018) presents a technology (UniDoSA) that is not exclusively focused on REST APIs, also supporting SOAP (Simple Object Access Protocol) and SCA (Service Component Architecture). Besides that, Palma, Moha and Guéhéneuc (2018) explains that to use UniDoSA with REST services it is necessary to first perform a semi-automatic conversion to the SCA technology. The Web prototype version of the tool does accept OpenAPI specifications directly, however it is not properly working as we already explained in the introduction of this chapter.

## 6.5 *Final remarks*

In this chapter, we presented Oasis, a proof of concept tool to detect anti-patterns in OpenAPI specifications. Oasis relies on static analysis techniques and requires no other input besides the specification of the API one wishes to analyze. As explained, our main motivation in creating it was the validation of OAS DB. In the next chapter, we show all the experiments that were carried out for the validation of our research.



## 7 Validation

Since our research resulted in contributions of different natures, we employed multiple strategies for validating our work. Section 7.1 discusses the validation of Oasis and of issues that OAS DB injects directly into generated OpenAPI specifications (these are detectable by using static analysis techniques, such as the ones implemented by Oasis). Section 7.2 presents the validation of OAS DB’s capability of injecting faults and issues in the mock API implementations it generates. These require the usage of tools that perform dynamic analysis on REST APIs. We devised and executed two experiments using the same set of mock APIs generated by OAS DB. In the first experiment, RESTler (presented in subsection 3.5.1) is used; for the second one, RESTest (summarized in subsection 3.5.2) is employed. Section 7.3 explains the validation of the novel REST anti-patterns introduced in chapter 4. Finally, section 7.4 presents and discusses threats to the validity of the aforementioned validation strategies.

### 7.1 *Validating OAS DB with tools employing static analysis*

As explained in chapter 6, our main motivation for creating Oasis was the fact that other similar tools mentioned in the literature were simply broken and could not be used to validate OAS DB. Obviously, then, the experiment designed to validate OAS DB by tools using static techniques relies on Oasis.

#### 7.1.1 Experiment settings

For the experiment, we used OAS DB to generate one complete OpenAPI specification for each seed (see section 5.1 for further detail on OpenAPI seeds) for each one of the anti-patterns detectable by Oasis (and that OAS DB is also able to inject into a specification). The seeds included with OAS DB (and used in this experiment) are: **incident\_response.json** (representing an incident reporting API), **payment.json** (representing a payments API) and **project\_management.json** (representing a project management API). A full listing of **incident\_response.json** is available at appendix A;

**payment.json** can be examined at appendix B; finally, **project\_management.json** is printed at appendix C. A total of nine unique OpenAPI specifications were generated.

Table 4 shows the number of instances of a given issue that were expected in a generated sample and how many Oasis was actually able to detect.

Table 4 – Experimental Evaluation Results - Oasis

Sample name	Issues	Expected	Detected
payment_amorphous_uri	amorphous_uri	4	4
project_management_sensitive_info_pqs	sensitive_info_pqs	1	1
project_management_amorphous_uri	amorphous_uri	4	4
incident_response_amorphous_uri	amorphous_uri	4	4
payment_crudy_uri	crudy_uri	4	1
incident_response_crudy_uri	crudy_uri	4	1
project_management_crudy_uri	crudy_uri	4	1
payment_sensitive_info_pqs	sensitive_info_pqs	1	1
incident_response_sensitive_info_pqs	sensitive_info_pqs	1	1

Source: Alex Braha Stoll, 2022

All the data produced during the experiment is available. It can be accessed at an anonymous public GitHub repository<sup>1</sup>.

### 7.1.2 Experiment discussion

Oasis had a satisfactory performance. Out of the nine samples of the experiment, it detected all instances of anti-patterns in six of them. In the three other samples, it failed to detect some of the anti-pattern instances. These samples were the ones that contained the Crudy URI anti-pattern: `payment_crudy_uri`, `incident_response_crudy_uri` and `project_management_crudy_uri`.

For these aforementioned three samples, Oasis detected only the cases of URIs having the *delete* verb in them (e.g., `delete_charges/{charge_id}`). It was not able to detect three remaining instances of the same anti-pattern in each of these three samples. These instances were the ones including a verb in the URI (and therefore being cases of the Crudy URI anti-pattern), however the verb included was not part of the list present in the rule that Oasis implements for the detection of this particular anti-pattern. Here, Oasis follows the guidelines given at Brabra *et al.* (2019).

<sup>1</sup> <https://github.com/oasdb/oasis-experiment>

### 7.1.3 Experiment conclusion

Despite its limitations, this experiment shows both **a)** OAS DB's capability of generating OpenAPI specifications including some of the anti-patterns discussed in the literature and **b)** Oasis satisfactory performance in detecting these same anti-patterns.

As shown and discussed in the previous section, OAS DB is able to generate specifications with instances of anti-patterns that are not detectable by Oasis. This indicates that the rules for detecting REST API anti-patterns presented in the relevant literature, themselves guidelines which Oasis implements, can be extended and improved.

## 7.2 *Validating OAS DB with tools employing dynamic analysis*

As discussed in chapter 5, OAS DB is also able to generate mock API implementations with known issues and faults. To detect those, it is necessary to interact with these running mock implementations. From the tools able to do so, we selected two that in our judgement are the most relevant and mature: RESTler (presented in subsection 3.5.1) and RESTest (discussed in subsection 3.5.2).

For both experiments, the same dataset was used. Using OAS DB's command-line interface (CLI), we generated one triad (specification, annotation file and mock API implementation) for each issue (that requires detection by dynamic techniques) that OAS DB is currently able to inject into synthesized assets (see subsection 5.3.2). This process was repeated for the same three different specification seeds mentioned in the previous section: `incident_response.json` (representing an incident reporting API), `payment.json` (representing a payments API) and `project_management.json` (representing a project management API). As a result, we ended up with 21 different generated triads.

As explained above, for this experiment we decided to generate assets containing only one issue each. It is important to note, however, that this is not a limitation of OAS DB itself. It is perfectly possible to feed to OAS DB's CLI a configuration file that will result in the generation of assets containing multiple of the issues currently supported by the tool.

### 7.2.1 RESTler

With the help of a custom Ruby script, we ran RESTler (version 7.3.0) for 15 minutes in its Fuzz mode for each of the mock API implementations generated by OAS DB. The results for each run were collected by our script and then manually analyzed by our team. These results are presented on table 5.

Table 5 – Experimental Evaluation Results - RESTler

Sample name	Issues	RESTler detected?
incident_response_payload_missing_keys	payload_missing_keys	yes
project_management_payload_extra_keys	payload_extra_keys	no
payment_broken_record_deletion	broken_record_deletion	yes
project_management_invalid_examples	invalid_examples	no
incident_response_unexpected_payload_root_node	unexpected_payload_root_node	yes
incident_response_payload_wrong_data_types	payload_wrong_data_types	yes
payment_unexpected_payload_root_node	unexpected_payload_root_node	yes
project_management_broken_record_deletion	broken_record_deletion	yes
payment_invalid_payload	invalid_payload	no
incident_response_broken_record_deletion	broken_record_deletion	yes
incident_response_payload_extra_keys	payload_extra_keys	no
project_management_unexpected_payload_root_node	unexpected_payload_root_node	yes
project_management_payload_missing_keys	payload_missing_keys	yes
project_management_payload_wrong_data_types	payload_wrong_data_types	no
payment_payload_missing_keys	payload_missing_keys	yes
payment_payload_wrong_data_types	payload_wrong_data_types	yes
payment_invalid_examples	invalid_examples	no
incident_response_invalid_examples	invalid_examples	no
project_management_invalid_payload	invalid_payload	no
payment_payload_extra_keys	payload_extra_keys	no
incident_response_invalid_payload	invalid_payload	no

Source: Alex Braha Stoll, 2022

All the data produced during the experiment is available. It can be accessed at an anonymous public GitHub repository<sup>2</sup>.

RESTler was able to detect 11 out of the 21 issues injected by OAS DB. Table 5 shows the detailed results for the aforementioned described experiment.

RESTler had a robust performance and was able to detect all **broken\_record\_deletion** issues (see subsection 5.3.2 for the list and discussion of injectable faults and issues). The tool was also able to detect most of the issues associated with an incorrect payload.

From the cases in which RESTler was not able to detect the issue present in the specification / API (10 instances), it is important to note that three of them involve

<sup>2</sup> <https://github.com/oasdb/restler-experiment>

the ExamplesChecker. From our analysis, it seems that RESTler is altogether failing to execute this checker. We believe this may be a bug in the tool.

The remaining seven cases that went undetected are all related to incorrect data payloads being sent to the server. Of these, three are cases of APIs having the **invalid\_payload** fault. The PayloadBodyChecker did not produce completely invalid JSON payloads (i.e., sending rubbish data that is not even a valid JSON) and therefore did not trigger this fault. Three other scenarios that went undetected are from generated APIs that have the **payload\_extra\_keys** issue. Since the PayloadBodyChecker did not generate payloads with extra and unexpected keys, this fault was not triggered.

Of these seven cases that were not detected, the remaining one is curious. RESTler failed to trigger the **payload\_wrong\_data\_type** only for the project\_management sample; for the other two experiment samples (incident\_response and payment), it was able to detect this same issue. Analyzing the logs, we noticed that RESTler did not attempt to generate a payload passing data other than strings in the case of the project\_management sample (the project\_management sample API expects a payload consisting exclusively of two string fields). For the cases in which an expected key is of a type different than string (e.g., an integer) — as it happens in the other mentioned samples — RESTler did generate payloads which had a wrong data type in these non-string keys, thus triggering the fault here being discussed. We believe this behavior may be a bug.

As a final note regarding the issues associated with incorrect data payloads, it is important to reiterate that we ran the PayloadBodyChecker with its default settings. Godefroid, Huang and Polishchuk (2020) discusses the workings and flexibility of this checker at length. We believe that it may be possible to tweak its configurations to a setting which would result in a better performance in the experiment herein discussed.

### 7.2.2 RESTest

With the aid of a script we wrote ourselves, we had RESTest (master branch at Git commit c0440ad81aa0d12b87732fdf05a82ddcafa74f6c) generate and run 224 test cases in its FT mode (fuzzing) for each of the mock API implementations generated by OAS DB. RESTest took about 20 minutes to test each sample. The results for each run were collected by our script and then manually analyzed by our team. These results are presented on

table 6. All the data produced during the experiment is available. It can be accessed at an anonymous public GitHub repository<sup>3</sup>.

Table 6 – Experimental Evaluation Results - RESTest

Sample name	Issues	RESTest detected?
incident_response_payload_missing_keys	payload_missing_keys	yes
project_management_payload_extra_keys	payload_extra_keys	no
payment_broken_record_deletion	broken_record_deletion	no
project_management_invalid_examples	invalid_examples	no
incident_response_unexpected_payload_root_node	unexpected_payload_root_node	no
incident_response_payload_wrong_data_types	payload_wrong_data_types	yes
payment_unexpected_payload_root_node	unexpected_payload_root_node	no
project_management_broken_record_deletion	broken_record_deletion	no
payment_invalid_payload	invalid_payload	no
incident_response_broken_record_deletion	broken_record_deletion	no
incident_response_payload_extra_keys	payload_extra_keys	no
project_management_unexpected_payload_root_node	unexpected_payload_root_node	no
project_management_payload_missing_keys	payload_missing_keys	yes
project_management_payload_wrong_data_types	payload_wrong_data_types	yes
payment_payload_missing_keys	payload_missing_keys	yes
payment_payload_wrong_data_types	payload_wrong_data_types	yes
payment_invalid_examples	invalid_examples	no
incident_response_invalid_examples	invalid_examples	no
project_management_invalid_payload	invalid_payload	no
payment_payload_extra_keys	payload_extra_keys	no
incident_response_invalid_payload	invalid_payload	no

Source: Alex Braha Stoll, 2022

Out of 21 faults and issues, RESTest was able to detect six of them. Although its performance was inferior to that of RESTler, we still consider RESTest to be a promising tool.

RESTest failed to detect four types of issues: **payload\_extra\_keys**, **broken\_record\_deletion**, **invalid\_examples** and **unexpected\_payload\_root\_node**. We follow with our analysis on the reasons the tool failed to detect each one of these types of faults.

To detect the **payload\_extra\_keys** issue, a tool must generate faulty payload that include keys that are not part of what is described by the mock API's corresponding OpenAPI specification. RESTest generators do not create tests of this type and, as a consequence, the tool is unable to trigger this error in the mock APIs.

To trigger the **broken\_record\_deletion** fault, the tool must be able to keep track of its interactions with the API and execute requests in a specific sequence: it needs to create a record, then to delete it at some point during the test suite and, finally, the tool needs

<sup>3</sup> <https://github.com/oasdb/restest-experiment>

to attempt to interact with the deleted object again (by trying to update or read it). From our analysis of the generated test cases, RESTest does not keep track of the IDs of created records, nor generates sequences of tests that engage in the interaction just described. As a result, it fails to detect the **broken\_record\_deletion** issue. RESTest’s documentation<sup>4</sup> mentions it also has stateful data generators, such as one they named BodyGenerator. This generator in particular is not available in the fuzzing mode, but as an extra exercise we did try to use it directly in a couple of *ad hoc* tests. Even when this specific generator was employed, the tool was not able to detect issues of type **broken\_record\_deletion**.

To expose **invalid\_examples** issues, a tool needs to attempt to use the payload examples provided as part of the OpenAPI specification when interacting with the mock API. Since RESTest does not do that, it fails to detect this kind of fault.

Finally, to catch **unexpected\_payload\_root\_node** faults, a tool needs to send to the mock API payloads that are completely malformed (e.g., an invalid JSON). RESTest does not attempt to do that and consequently it is not able to detect the fault herein discussed.

### 7.2.3 Experiment conclusion

By testing real-world tools against OAS DB generated mock APIs, we have shown that our approach has potential. By using OAS DB capabilities, one is able to synthesize a great number of varying scenarios, probably far greater than what would be feasible to be manually put together.

We are confident that the results of the experiment support the case for OAS DB and its approach. We were able to find two potential bugs in the current implementation of RESTler. Besides that, we were able to uncover many different improvements both RESTler and RESTest can incorporate to improve their capabilities and make the tools even more apt in detecting faults and issues in REST APIs.

## 7.3 Novel REST anti-patterns

Since the contribution of new REST anti-patterns (see chapter 4) is a purely theoretical one, validating every new REST anti-pattern proposed means justifying their

<sup>4</sup> <https://github.com/isa-group/RESTest/blob/master/README.md>

relevance by showing that their presence in a REST API may negatively impact the system in at least one dimension (e.g., security). Another source of validation is the connection of the anti-pattern with previous studies and reliable industry reports. In the case of anti-patterns that may result in vulnerabilities in the affected system, for example, linking the anti-pattern to one or more CVEs (Common Vulnerabilities and Exposures) is a strong indicator that our proposal in fact exposes a security issue already recognized by the software community at large.

As one can check by exploring chapter 4, we already justified the relevance of each proposed anti-pattern together with its definition and detailed explanation. Subsection 4.1.1 presents and validates the anti-pattern *Sequential integers as resource ID*; subsection 4.2.1 discusses in depth the anti-pattern *Sensitive information in the path or in the query string* (already catalogued in the literature) and offers a solution to APIs showing this issue.

## 7.4 Threats to validity

### 7.4.1 External validity

The samples generated by OAS DB can be seen as proxies of real-world APIs, having in mind the context of validation of tools that detect anti-patterns and faults in REST APIs. This is the case because when building OAS DB — as discussed throughout our research — we aimed at reproducing anti-patterns and faults that are already discussed and catalogued in the relevant literature (which in turn is based upon, at least in part, observations of real-world systems). The experiments that we ran corroborate this argument, since tools that are capable of finding anti-patterns and faults in real-world APIs are also able to detect these same problems in the assets generated by OAS DB.

### 7.4.2 Internal validity

We argue that both our experiments are internally valid and that they show a clear cause and effect relationship. To guarantee that, for both experiments we conducted a detailed analysis of the results.



Through this analysis, we were able to explain why each tool (Oasis, RESTler and RESTest) failed to detect a given expected anti-pattern or fault. When we were not able to identify the cause of a failure of detection, we clearly indicated that and even in some cases were able to come up with a hypothesis on why the failure happened (e.g. because of a bug in the tool).

#### 7.4.3 Conclusion validity

By testing real-world tools against OAS DB generated specifications and mock APIs, we have shown that our approach has potential.

At least for the types of anti-patterns and issues that are part of the scope of the present research and of the experiments in question, we can conclude that generating specifications and mock API implementations can indeed help surface weaknesses of tools aiming to test REST APIs. As a result of that, we were able to demonstrate that OAS DB may prove useful to research in the field by helping to make these and other tools more precise and capable.

#### 7.5 *Final remarks*

In this chapter, we discussed all methods we used to validate all the fruits that the present research bears. We also deliberated on the reasons why our chosen methods are indeed valid.

We carefully examined all experiments made to validate OAS DB, using tools that rely on static and dynamic analysis techniques. Additionally, we touched on the validation of the novel anti-pattern we introduced, which is the only purely theoretical contribution of our work. We believe that the tools and techniques validated in this chapter can become an important resource in helping researchers in the field of REST API and REST specifications.

## 8 Conclusion

By reviewing the literature, we were able to notice that currently there are not that many studies focusing on finding bugs and violations of REST API best practices. The necessity of further investigating issues that are found in REST APIs is even directly pointed out by some of the reviewed works (e.g., Atlidakis, Godefroid and Polishchuk (2019)). Furthermore, we identified that there are no widely accepted benchmarks for evaluating OpenAPI related tools. As a consequence, researchers have to spend time creating their own datasets for validation purposes and the comparison between different works becomes harder.

Having acquired in depth knowledge of the anti-patterns described in the literature, we identified an opportunity to describe a new REST anti-pattern that — to the best of our knowledge — is not yet documented. Chapter 4 describes the *Sequential integers as resource ID* novel anti-pattern. In summary, it happens when the resources of an API are identified by sequential integer numbers. As explained before, this may increase the chances of an attacker being able to successfully exploit a system and may also lead to undesirable business information leakage.

To address the lack of a widely accepted benchmark in the OpenAPI research community, we created OAS DB. As shown in chapter 5, our approach and implementation resulted in a tool capable of generating annotated OpenAPI specifications and their corresponding mock API implementations. OAS DB is capable — during this generation process — to inject a series of different anti-patterns and faults, both in the OpenAPI specification and in its implementation itself.

To supplement the validation of OAS DB, we saw ourselves with no options but to implement a proof of concept tool to detect anti-patterns in REST APIs by means of static analysis of their OpenAPI specifications. Chapter 6 goes through the motivation for creating Oasis and also explains in detail how the tool works.

Chapter 7 presents all the strategies and experiments done to validate our research. For validating OAS DB, for example, we did three separate experiments, one using Oasis and the other two using two mature dynamic tools, RESTler and RESTest. We were able to show that OAS DB can be a useful resource to help support the evolution of OpenAPI related tools.

In the sections below, we enumerate and further discuss the contributions that are the fruits of the present research. Then, we indicate some possibilities for continuing the work we did and further improving OAS DB.

## 8.1 Contributions

### 8.1.1 Proposal of a new REST anti-pattern

As a consequence of studying the relevant literature in depth and combining the acquired knowledge with our industry experience, we were able to identify an anti-pattern that occurs in real-world APIs but is still not documented. Chapter 4 proposes the *Sequential integers as resource ID* new REST anti-pattern. There, we described the anti-pattern and all its negative consequences, ranging from leakage of important business data to security vulnerabilities.

### 8.1.2 OAS DB

OAS DB is our main contribution. It is our take on how to address the challenge of the lack of a standard repository of OpenAPI samples. Since there is no widely accepted dataset, each tool developer comes up with its own. As a consequence, researchers have to dedicate time they could spend improving the core of their work to building validation datasets. This lack of a standard collection of samples also makes it very challenging to compare different tools and to identify their strenghts and weaknesses.

As chapter 5 shows, OAS DB is much more than a dataset. It is actually a generator of samples. Furthermore, OAS DB produces machine-readable files (which we call *annotation files*) that indicate all issues to be found in a generated OpenAPI sample. OAS DB is also able to synthesize mock implementations of the generated OpenAPI samples. By having all these capabilities, OAS DB can be used by both researchers of tools employing static and dynamic techniques. We believe OAS DB – and more broadly the *approach* we pioneered by developing OAS DB – may prove of great value to the REST API / OpenAPI research community.

### 8.1.3 Oasis

As we make clear in chapter 6, our main motivation in creating Oasis was to also be able to validate OAS DB with a tool employing static analysis techniques only. Nonetheless, the result *is* a tool that already implements a few REST anti-patterns described in the literature and that can be extended to be able to detect other anti-patterns and issues.

### 8.1.4 Future work

We argue that the most promising opportunities involve expanding OAS DB (or a successor project leveraging the same approach). Here are some lines of research we believe have potential:

1. To expand OAS DB, making it capable of injecting many more types of anti-patterns, issues and faults, both into OpenAPI specifications and their corresponding mock API implementations;
2. To explore the idea of creating OpenAPI specification seeds from existing real-world OpenAPI specifications, by automatically simplifying them and extracting their essentials. These seeds can then be fed to OAS DB as manually created ones are today. This technique could produce more realistic seeds and allow one to generate a vast number of seeds with virtually no manual effort;
3. To evolve the OpenAPI seed format, allowing one to express more complexity with it and therefore making it possible to improve OAS DB so it can generate richer OpenAPI specifications and even more interesting mock API implementations;
4. To conduct more experiments using both tools based on static analysis and on dynamic analysis.

## Bibliography

- ALONSO, J. C.; MARTIN-LOPEZ, A.; SEGURA, S.; GARCIA, J. M.; RUIZ-CORTES, A. Arte: Automated generation of realistic test inputs for web apis. *IEEE Transactions on Software Engineering*, p. 1–1, 2022. ISSN 1939-3520. Cited at page 42.
- ATLIDAKIS, V.; GODEFROID, P.; POLISHCHUK, M. RESTler: Stateful REST API fuzzing. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada: [s.n.], 2019. p. 748–758. ISSN 1558-1225. Cited 8 times at pages 15, 16, 38, 41, 42, 43, 53, and 81.
- BANIAŞ, O.; FLOREA, D.; GYALAI, R.; CURIAC, D.-I. Automated specification-based testing of rest apis. *Sensors*, v. 21, n. 16, 2021. Cited at page 42.
- BRABRA, H.; MTIBAA, A.; PETRILLO, F.; MERLE, P.; SLIMAN, L.; MOHA, N.; GAALOUL, W.; GUÉHÉNEUC, Y.-G.; BENATALLAH, B.; GARGOURI, F. On semantic detection of cloud API (anti)patterns. *Information and Software Technology*, Elsevier B.V., v. 107, p. 65–82, 2019. Cited 8 times at pages 24, 25, 35, 53, 67, 70, 71, and 73.
- CORRADINI, D.; ZAMPIERI, A.; PASQUA, M.; VIGLIANISI, E.; DALLAGO, M.; CECCATO, M. Automated black-box testing of nominal and error scenarios in restful apis. *Software Testing Verification and Reliability*, 2022. Cited at page 42.
- CVE-2015-8542. 2015. Available from MITRE, CVE-ID CVE-2015-8542. Available from Internet: <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8542>. Cited at page 47.
- DO, H.; ELBAUM, S. G.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, v. 10, n. 4, p. 405–435, 2005. Available from Internet: <https://doi.org/10.1007/s10664-005-3861-2>. Cited 2 times at pages 18 and 44.
- ED-DOUBI, H.; IZQUIERDO, J. L. C.; CABOT, J. 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: [s.n.], 2018. p. 181–190. Cited 6 times at pages 37, 41, 53, 66, 67, and 71.
- FIELDING, R. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Cited 5 times at pages 15, 21, 22, 23, and 24.
- GODEFROID, P.; HUANG, B.; POLISHCHUK, M. Intelligent REST API data fuzzing. In: *2020 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC / FSE)*. Sacramento, USA: [s.n.], 2020. Cited 2 times at pages 42 and 76.
- GODEFROID, P.; LEHMANN, D.; POLISHCHUK, M. Differential regression testing for REST APIs. In: . [S.l.: s.n.], 2020. p. 312–323. Cited at page 42.

ISO 25010. 2011. Available from ISO, ISO 25010:2011. Available from Internet: <https://www.iso.org/standard/35733.html>. Cited at page 54.

IVERSEN, P. *Specification-based security analysis of REST APIs*. Dissertação (Mestrado) — Norwegian University of Science and Technology, 2018. Cited 7 times at pages 33, 34, 37, 41, 49, 53, and 70.

JUST, R.; JALALI, D.; ERNST, M. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*. [s.n.], 2014. p. 437–440. Available from Internet: <https://doi.org/10.1145/2610384.2628055>. Cited 2 times at pages 18 and 44.

KARLSSON, S.; CAUSEVIC, A.; SUNDMARK, D. Quickrest: Property-based test generation of OpenAPI-Described RESTful APIs. In: . [S.l.: s.n.], 2020. p. 131–141. Cited at page 43.

KISTOWSKI, J. von; EISMANN, S.; SCHMITT, N.; BAUER, A.; GROHMANN, J.; KOUNEV, S. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. [S.l.: s.n.], 2018. (MASCOTS '18). Cited at page 43.

MARTIN-LOPEZ, A.; SEGURA, S.; RUIZ-CORTÉS, A. RESTest: Automated Black-Box Testing of RESTful Web APIs. In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. [S.l.]: Association for Computing Machinery, 2021. (ISSTA '21). Cited 3 times at pages 39, 40, and 42.

MAXIMILIEN, E. M.; RANABAHU, A.; GOMADAM, K. An online platform for Web APIs and service mashups. *IEEE Internet Computing*, v. 12, n. 5, p. 32–43, 2008. Cited at page 25.

MIRABELLA, A. G.; MARTIN-LOPEZ, A.; SEGURA, S.; VALENCIA-CABRERA, L.; RUIZ-CORTES, A. Deep learning-based prediction of test input validity for RESTful APIs. In: . [S.l.: s.n.], 2021. p. 9–16. Cited at page 43.

Modelling Languages Blog Post. 2018. Available from Modelling Languages Blog. Available from Internet: <https://modeling-languages.com/automatic-generation-of-test-cases-for-rest-apis/>. Cited at page 67.

Official Docker Website. 2020. Available from Docker Inc. Available from Internet: <https://www.docker.com/resources/what-container>. Cited 2 times at pages 27 and 28.

Official Raygun Blog. 2020. Available from Raygun. Available from Internet: <https://raygun.com/blog/soap-vs-rest-vs-json/>. Cited at page 21.

OpenAPI version 3.0.3. 2020. Available from OpenAPI Initiative, OpenAPI version 3.0.3. Available from Internet: <https://spec.openapis.org/oas/v3.0.3>. Cited at page 52.

PALMA, F.; MOHA, N.; GUÉHÉNEUC, Y. UniDoSA: The unified specification and detection of service antipatterns. *IEEE Transactions on Software Engineering*, 2018. Cited 4 times at pages 36, 53, 70, and 71.

PETRILLO, F.; MERLE, P.; MOHA, N.; GUÉHÉNEUC, Y.-G. Are REST APIs for cloud computing well-designed? an exploratory study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer Verlag, v. 9936 LNCS, p. 157–170, 2016. Cited 5 times at pages 16, 32, 33, 45, and 53.

RFC 2818. 2000. Available from IETF Trust, RFC 2818. Available from Internet: [⟨https://tools.ietf.org/html/rfc2818⟩](https://tools.ietf.org/html/rfc2818). Cited at page 19.

RFC 7231. 2014. Available from IETF Trust, RFC 7231. Available from Internet: [⟨https://tools.ietf.org/html/rfc7231⟩](https://tools.ietf.org/html/rfc7231). Cited 5 times at pages 19, 22, 23, 25, and 57.

RFC 7519. 2015. Available from IETF Trust, RFC 7519. Available from Internet: [⟨https://tools.ietf.org/html/rfc7519⟩](https://tools.ietf.org/html/rfc7519). Cited at page 34.

RFC 8446. 2018. Available from IETF Trust, RFC 8446. Available from Internet: [⟨https://tools.ietf.org/html/rfc8446⟩](https://tools.ietf.org/html/rfc8446). Cited at page 19.

SIMBAD Tool Blog Post. 2019. Available from SIMBAD authors. Available from Internet: [⟨http://www-inf.it-sudparis.eu/SIMBAD/tools/ORAP-Detection/⟩](http://www-inf.it-sudparis.eu/SIMBAD/tools/ORAP-Detection/). Cited at page 67.

SOAP Draft. 2001. Available from W3C. Available from Internet: [⟨https://www.w3.org/TR/2001/WD-soap12-20010709/⟩](https://www.w3.org/TR/2001/WD-soap12-20010709/). Cited at page 21.

SOAP Specification. 2007. Available from W3C. Available from Internet: [⟨https://www.w3.org/TR/soap12-part1/⟩](https://www.w3.org/TR/soap12-part1/). Cited at page 20.

# Appendix



## APPENDIX A – incident\_response.json OpenAPI sample seed

```
{
  "info": {
    "title": "Incident response API",
    "description": "API that allows one to report and manage
      incidents that happen in an online service (e.g. an
      ecommerce).",
    "version": "0.1"
  },
  "components": {
    "schemas": {
      "Incident": {
        "type": "object",
        "description": "An incident.",
        "properties": {
          "title": {
            "description": "A title to identify the incident
              .",
            "type": "string"
          },
          "service_id": {
            "description": "The ID of the affected service.",
            "type": "integer"
          },
          "assignee_id": {
            "description": "The ID of the colaborator
              assigned to deal with the issue.",
            "type": "integer"
          }
        }
      },
      "example": {
```

```
    "title": "30 minutes outage due to someone tripping
              on the server's power cable.",
    "service_id": 101,
    "assignee_id": 11
  }
}
}
}
```

## APPENDIX B – payment.json OpenAPI sample seed

```
{
  "info": {
    "title": "Payments Provider API.",
    "description": "API that allows one to charge customers
      via credit card.",
    "version": "0.1"
  },
  "components": {
    "schemas": {
      "Charge": {
        "type": "object",
        "description": "A charge.",
        "properties": {
          "amount": {
            "description": "The amount to be charged.",
            "type": "number"
          },
          "currency": {
            "description": "Three-letter ISO currency code.",
            "type": "string"
          },
          "credit_card_id": {
            "description": "The credit card to be charged.",
            "type": "integer"
          }
        }
      },
      "example": {
        "amount": 20.50,
        "currency": "USD",
        "credit_card_id": 1
      }
    }
  }
}
```

```
}  
}  
}  
}  
}
```

## APPENDIX C – project\_management.json OpenAPI sample seed

```
{
  "info": {
    "title": "Project management API",
    "description": "API that allows one to manage ongoing
      projects in a company (e.g., create / read / update /
      delete projects).",
    "version": "0.1"
  },
  "components": {
    "schemas": {
      "Project": {
        "type": "object",
        "description": "A project.",
        "properties": {
          "name": {
            "description": "The name of the project.",
            "type": "string"
          },
          "description": {
            "description": "A meaningful description of the
              project.",
            "type": "string"
          }
        },
        "example": {
          "name": "Client A",
          "description": "An awesome project for Client A."
        }
      }
    }
  }
}
```

}

}

## APPENDIX D – Anti-patterns and issues that OAS DB is capable of injecting during asset generation

Name	Config. issue list	Config. value	Description
Crudy URI	spec.issues	crudy_uri	Injects a HTTP action verb (or a synonym) in the URL
Amorphous URI	spec.issues	amorphous_uri	Adds superfluous characters to the URL (e.g. a file type suffix such as '.xml')
Ignoring status code	spec.issues	ignoring_status_code	Responses use inappropriate HTTP codes
Inappropriate HTTP method	spec.issues	inappropriate_http_method	Requests expect inappropriate HTTP methods
Invalid examples	spec.issues	invalid_examples	Generates request examples that do not comply to the related objects' schema
Sensitive info in the pqs	spec.issues	sensitive_info_pqs	Sensitive info (e.g. a token) is included in paths or query strings
Invalid payload	api.issues	invalid_payload	API crashes (500 response) if an invalid payload is received
Unexpected payload root node	api.issues	unexpected_payload_root_node	API crashes if the root node of the received payload is not an object
Payload missing keys	api.issues	payload_missing_keys	API crashes if the received payload misses expected keys
Payload extra keys	api.issues	payload_extra_keys	API crashes if the received payload has extra keys
Payload wrong data types	api.issues	payload_wrong_data_types	API crashes if the received payload has fields with data that does not comply to the data types specified in the object's schema inside the correspondent OpenAPI spec
Broken record deletion	api.issues	broken_record_deletion	API responds as if an object was successfully deleted, but the object is not actually destroyed

Source: Alex Braha Stoll, 2022









