

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Stateful Security Testing of Web APIs

Gédsimon Soares Pereira



Mestrado em Engenharia de Software

Supervisor: Nuno Macedo

Second Supervisor: Hugo Pacheco

October 17, 2024



# **Stateful Security Testing of Web APIs**

**Gédsimon Soares Pereira**

Mestrado em Engenharia de Software

Approved in oral examination by the committee:

President: Prof. Doutor Nuno Honório Rodrigues Flores, Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto,

Referee: Prof. Doutor João Marco Cardoso Silva, Professor Auxiliar da Universidade do Minho

Referee: Prof. Doutor Nuno Filipe Moreira Macedo, Professor Auxiliar da Faculdade de Engenharia da Universidade do Porto

October 17, 2024

# Resumo

Os modelos de arquitetura, como os microsserviços, evoluíram para facilitar a interligação entre aplicações. Em aplicações Web mais modernas utilizam API RESTful (Application Programming Interface) para permitir a comunicação entre serviços e a especificação OpenAPI para descrever estes serviços de uma forma que possam ser interpretados por humanos e máquinas.

A utilização de API tem sido cada vez mais comum, em relatórios emitidos pela Postman, que é uma plataforma utilizada por desenvolvedores de APIs mundialmente, nos últimos 3 anos, tem visto crescimentos significativos a cada ano na utilização da plataforma, apresentando um crescimento de 15 milhões de utilizadores comparado com os números de 2022, quando reportou ter 20 milhões de utilizadores ativos, o que traduz num número significativo de APIs criadas ao longo dos anos.

Um questão que nos colocamos enquanto investigadores, programadores e engenheiros de segurança é a de saber até que ponto todas estas APIs são seguras e fiáveis. Por isso, para responder a essa questão, precisamos realmente de automação para suportar esta escala de crescimento.

Nos últimos anos, engenheiros de software têm adotado ferramentas e técnicas para ajudar a atenuar este problema e garantir que o processo de teste de segurança possa abranger vários aspectos do sistema, como disponibilidade, integridade, confidencialidade e a conformidade.

Operações comuns como `PUT` e `DELETE` devem ser idempotentes, o que significa que o envio do mesmo pedido mais de uma vez deve ter o mesmo resultado. Os resultados destas actividades podem ser inconsistentes, a menos que sejam testados exaustivamente. Neste sentido, as ferramentas de geração de testes automatizados desempenham um papel importante, na medida em que podem criar um maior número de casos de teste do que a criação manual de testes.

Por exemplo, um `DELETE` deve seguir-se logicamente a um `POST`, uma vez que remove recursos que foi criado pela operação `POST`. No entanto, podem ocorrer erros quando a sequência não é gerida cuidadosamente, levando a tentativas de eliminação de recursos que podem ainda não existir ou ter dependências totalmente estabelecidas. Muitas APIs envolvem recursos interdependentes, em que a ordem correta das operações é essencial para garantir a integridade dos dados e evitar problemas como entradas órfãs, relações quebradas ou perda involuntária de dados.

A tentativa de utilizar métodos para os quais os endpoints não estão explicitamente destinados, por exemplo, enviar um pedido `GET` para alterar dados ou enviar um pedido `POST` para obter dados, pode resultar em comportamentos inesperados ou em funcionalidades inexistentes. Por exemplo, sem verificações rigorosas do método no endpoint, um pedido `GET` poderia inadvertidamente desencadear modificações, levando a registos inexistentes ou imprecisos, uma vez que não seria gerado qualquer registo de alteração adequado.

Portanto, este trabalho propõe uma extensão para o OpenAPI para especificar relações entre pedidos de forma a poder possibilitar ferramentas de geração de testes automáticos a gerar testes com mais precisão, também tem como objetivo estender a ferramenta RESTler para suportar testes de segurança baseados em propriedades com base em informações extraídas da especificação (OpenAPI) e pedidos.

**Palavras-chave:** restful, webapi, property-based testing, openapi, fuzzing, automated testing, black box, security testing, grammar testing

# Abstract

Architectural models, such as microservices, have evolved to facilitate interconnected applications. More modern web applications use RESTful APIs (Application Programming Interface) to enable communication between services and *OpenAPI Specification* to describe these services in a way that humans and machines can interpret.

APIs are increasingly used, in reports published by Postman, which is a platform used by API developers worldwide, over the last 3 years, has seen significant growth each year in the use of the platform, showing a growth of 15 million users in 2024 compared to 2022, when it reported having 20 million active users, which results in a significant number of APIs created over the years.

As researchers, programmers and security engineers, we ask ourselves to what extent all these APIs are secure and reliable. So, to answer that question, we need automation to support this scale of growth.

Software engineers should follow several recommendations and standards, as well as a checklist<sup>1</sup>, to support the design and implementation of security-related Web APIs. These APIs should be built in such a way as not to jeopardise the information that is exchanged between entities.

In recent years, developers have adopted tools and techniques to help alleviate this problem and ensure that the security testing process can cover various aspects of the system, such as availability, integrity, confidentiality, and compliance.

Common operations like `PUT` and `DELETE` should be idempotent, meaning that sending the same request more than once should have the same result. The results of these activities may be inconsistent unless they are tested thoroughly. In this sense, automated test generation tools play an important role in that they can generate a greater number of test cases than manual test generation.

For instance, a `DELETE` request should logically follow a `POST` request, as it removes resources that the `POST` created. However, errors can occur when the sequence is not carefully managed, leading to attempts to delete resources that may not yet exist or have fully established dependencies. Many APIs involve interdependent resources, where the proper order of operations is essential to ensure data integrity and prevent issues like orphaned entries, broken relationships, or unintended data loss.

Attempting to use methods for which endpoints are not explicitly meant, for example, sending a `GET` request to change data or sending a `POST` request to fetch data, can result in unusual behaviours or non-existent features. For instance, without strict method checks at the endpoint, a `GET` request could inadvertently trigger modifications, leading to missing or inaccurate logs, as no proper change record would be generated.

Thus, this work proposes an extension to OpenAPI to specify relationships between requests, enabling automated test generation tools to generate tests more accurately. It also aims to extend the RESTler tool to support property-based security tests based on information extracted from the specification (OpenAPI) and requests.

---

<sup>1</sup><https://github.com/shieldfy/API-Security-Checklist>

**Keywords:** restful, webapi, property-based testing, openapi, fuzzing, automated testing, black box, security testing, grammar testing

# Acknowledgements

Firstly, I would like to thank my family, those who are close by, my children and wife, those who are far away, my siblings and mother, and my father in memory, who have supported me throughout this journey that has not been easy, without your support I would certainly not have made it this far. I would also like to thank my course mates who participated in this journey. Thank you for your partnership and support when I thought handing in my work on time was impossible.

To my supervisors, Professors Hugo Pacheco and Nuno Macedo, for their tireless support and motivation, even at times when the goal seemed difficult to achieve, they were always available to meet and discuss, to find the best solution to the problem we set out to solve.

Gédsimon Soares Pereira

*“Intelligence is the ability to adapt to change.”*

Stephen Hawking

# Contents

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| 1.1      | Problem Definition . . . . .                      | 2         |
| 1.2      | Objectives . . . . .                              | 2         |
| 1.3      | Contribution . . . . .                            | 3         |
| 1.4      | Dissertation Structure . . . . .                  | 3         |
| <b>2</b> | <b>Background</b>                                 | <b>4</b>  |
| 2.1      | REST and RFC 9110 . . . . .                       | 4         |
| 2.2      | OpenAPI Specification . . . . .                   | 5         |
| 2.3      | Fuzzing Testing . . . . .                         | 7         |
| 2.4      | Maturity Models in Detail . . . . .               | 8         |
| 2.4.1    | Level 0 - Swamp of POX . . . . .                  | 9         |
| 2.4.2    | Level 1 - Resources . . . . .                     | 9         |
| 2.4.3    | Level 2 - HTTP Verbs . . . . .                    | 10        |
| 2.4.4    | Level 3 - Hypermedia Controls (HATEOAS) . . . . . | 10        |
| <b>3</b> | <b>Literature Review</b>                          | <b>14</b> |
| 3.1      | Research Approach . . . . .                       | 14        |
| 3.1.1    | Database and Search Queries . . . . .             | 14        |
| 3.1.2    | Paper Selection Criteria . . . . .                | 15        |
| 3.2      | Tools . . . . .                                   | 15        |
| 3.2.1    | RESTler . . . . .                                 | 15        |
| 3.2.2    | Schemathesis . . . . .                            | 17        |
| 3.2.3    | RESTest . . . . .                                 | 17        |
| 3.2.4    | GADOLINIUM . . . . .                              | 18        |
| 3.2.5    | foREST . . . . .                                  | 20        |
| <b>4</b> | <b>RESTler</b>                                    | <b>24</b> |
| 4.1      | Introduction . . . . .                            | 24        |
| 4.2      | RESTler in Detail . . . . .                       | 25        |
| 4.2.1    | RESTler Compiler . . . . .                        | 25        |
| 4.2.2    | RESTler Test . . . . .                            | 26        |
| 4.2.3    | RESTler Fuzz-Lean . . . . .                       | 28        |
| 4.2.4    | RESTler Fuzz . . . . .                            | 29        |
| 4.3      | Extending RESTler . . . . .                       | 29        |

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| <b>5</b> | <b>Design and Implementation</b>                              | <b>31</b> |
| 5.1      | Current Case Study . . . . .                                  | 32        |
| 5.2      | Design . . . . .                                              | 33        |
| 5.3      | OpenAPI Extensions for Related Operations . . . . .           | 36        |
| 5.4      | Implementation of RESTler Checkers . . . . .                  | 39        |
| 5.4.1    | Sequence Checker . . . . .                                    | 39        |
| 5.4.2    | Dependencies-Hierarchy Checker . . . . .                      | 43        |
| 5.4.3    | Idempotence Checker . . . . .                                 | 46        |
| 5.4.4    | Method Not Allowed Checker - MnA . . . . .                    | 49        |
| <b>6</b> | <b>Evaluation</b>                                             | <b>52</b> |
| 6.1      | Evaluation Process . . . . .                                  | 52        |
| 6.1.1    | Test Environment Setup . . . . .                              | 52        |
| 6.1.2    | Metrics Collected During Validation . . . . .                 | 57        |
| 6.2      | Discussion . . . . .                                          | 61        |
| 6.3      | Limitations . . . . .                                         | 63        |
| <b>7</b> | <b>Conclusions and Future Work</b>                            | <b>64</b> |
| 7.1      | Conclusions . . . . .                                         | 64        |
| 7.2      | Future Works . . . . .                                        | 64        |
|          | <b>References</b>                                             | <b>66</b> |
| <b>A</b> | <b>List of Publications Included in the Literature Review</b> | <b>68</b> |
| <b>B</b> | <b>RESTler and Application Files</b>                          | <b>71</b> |
| B.1      | Dictionary . . . . .                                          | 71        |
| B.2      | Spec Coverage . . . . .                                       | 71        |
| B.3      | RESTler Sequence . . . . .                                    | 71        |
| B.4      | BlogPost OAS . . . . .                                        | 71        |

# List of Figures

|     |                                                      |    |
|-----|------------------------------------------------------|----|
| 2.1 | HTTP Status Code . . . . .                           | 5  |
| 2.2 | Links in Swagger Spec . . . . .                      | 7  |
| 2.3 | Richardson Maturity model . . . . .                  | 8  |
| 3.1 | IEEE Command Query . . . . .                         | 14 |
| 3.2 | RESTler Architecture . . . . .                       | 15 |
| 3.3 | REStest workflow . . . . .                           | 18 |
| 3.4 | The workflow of proposed security analyses . . . . . | 19 |
| 3.5 | Framework of foREST . . . . .                        | 21 |
| 3.6 | Sample bug detected by foREST . . . . .              | 21 |
| 4.1 | RESTler Main Components . . . . .                    | 25 |
| 4.2 | RESTler Compiler fragments . . . . .                 | 26 |
| 4.3 | RESTler Test output . . . . .                        | 28 |
| 4.4 | RESTler Fuzz-Lean output . . . . .                   | 28 |
| 5.1 | RESTler with Extended Components . . . . .           | 31 |
| 5.2 | C4 Blog Post system context diagram . . . . .        | 32 |
| 5.3 | Request Sequence Fragment . . . . .                  | 35 |
| 5.4 | State Transition Valid Sequence . . . . .            | 40 |
| 5.5 | State Machine Invalid Sequences . . . . .            | 41 |
| 5.6 | Related Operations Fragment . . . . .                | 45 |
| 6.1 | Pet Store OpenAPI . . . . .                          | 54 |
| 6.2 | Failures Reported by Schemathesis . . . . .          | 58 |
| 6.3 | Resource Consumption . . . . .                       | 59 |
| 6.4 | Coverage Reported by Schemathesis . . . . .          | 59 |
| 6.5 | Sequence Checker - Bug Reported . . . . .            | 62 |
| B.1 | RESTler Dictionary . . . . .                         | 72 |
| B.2 | Request Sequence . . . . .                           | 74 |
| B.3 | BlogPost OAS . . . . .                               | 75 |

# List of Tables

|     |                                                                |    |
|-----|----------------------------------------------------------------|----|
| 5.1 | Analysing the Sequence Checker execution . . . . .             | 43 |
| 5.2 | HTTP Methods Idempotence and safeness classification . . . . . | 46 |
| 5.3 | HTTP Method Composition . . . . .                              | 51 |
| 5.4 | Analysing the MnA checker execution . . . . .                  | 51 |
| 6.1 | Issues Reported . . . . .                                      | 57 |
| 6.2 | Status Code Observed . . . . .                                 | 60 |
| A.1 | List of Publications . . . . .                                 | 68 |

# List of source codes

|    |                                                   |    |
|----|---------------------------------------------------|----|
| 1  | OpenAPI schema for /auth/register . . . . .       | 6  |
| 2  | RMM0 Request . . . . .                            | 9  |
| 3  | RMM0 Response . . . . .                           | 9  |
| 4  | RMM1 Request . . . . .                            | 11 |
| 5  | RMM1 Response . . . . .                           | 11 |
| 6  | RMM2 Request . . . . .                            | 12 |
| 7  | RMM2 Response . . . . .                           | 12 |
| 8  | RMM3 Request . . . . .                            | 13 |
| 9  | RMM3 Response . . . . .                           | 13 |
| 10 | RESTler Compiler . . . . .                        | 26 |
| 11 | RESTler Test . . . . .                            | 27 |
| 12 | RESTler Request Sample . . . . .                  | 27 |
| 13 | RESTler Response Sample . . . . .                 | 27 |
| 14 | RESTler Fuzz-Lean . . . . .                       | 28 |
| 15 | RESTler Fuzz . . . . .                            | 29 |
| 16 | Checker Base Fragment . . . . .                   | 30 |
| 17 | Request-Response Sequence Graph . . . . .         | 35 |
| 18 | OAS Extension x-related-operations . . . . .      | 39 |
| 19 | Sequence Checker Initial Validation . . . . .     | 41 |
| 20 | Sequence Checker Output . . . . .                 | 42 |
| 21 | Sequence Checker Requests . . . . .               | 42 |
| 22 | Idempotence Checker Initial Setup . . . . .       | 44 |
| 23 | Idempotence Checker Initial Setup . . . . .       | 47 |
| 24 | Idempotence Requests . . . . .                    | 48 |
| 25 | Idempotence Validate Response . . . . .           | 49 |
| 26 | Idempotence Report Bug . . . . .                  | 49 |
| 27 | RESTler Specification Coverage Fragment . . . . . | 73 |
| 28 | OpenAPI schema for BlogPosts . . . . .            | 76 |

# Abbreviations

|      |                                   |
|------|-----------------------------------|
| API  | Application Programming Interface |
| REST | Representational State Transfer   |
| JSON | JavaScript Object Notation        |
| YAML | Yaml Ain't Markup Language        |
| XML  | Extensible Markup Language        |
| POX  | Plain old XML                     |
| OAS  | OpenAPI Specification             |
| RFC  | Request for Comments              |
| WWW  | <i>World Wide Web</i>             |
| CLI  | command-line interface            |
| CRUD | Create, Read, Update, and Delete  |
| M2M  | Machine-to-Machine                |
| URI  | Uniform Resource Identifier       |
| URL  | Uniform Resource Locator          |

# Chapter 1

## Introduction

With the evolution of systems and architectural models, such as microservices, and the increasing need to exchange information between entities, it has become common for applications to be interconnected, whether internally or externally, and one of the most popular ways of exchanging information is through API.

The use of APIs is increasing these days, an example of which is a report released by Postman<sup>1</sup>, which reveals that in 2023, there were more than 25 million users on the platform, up from 20 million in 2022, resulting in the creation of over 121 million collections against 38 million in 2022, which translated into 1.29 billion requests created in the previous year, Portugal accounted for 6.69 million of these requests.

The common API standard used by most mobile and web applications to establish server communication is REST (Representational State Transfer), introduced by Roy Thomas Fielding in 2000 [6].

REST is not classified as a specification or a protocol but is a set of guidelines<sup>2</sup> which has become a standard for building web APIs, an API that follows the REST standard is called a RESTful<sup>3</sup> API, which is usually described using the OpenAPI Specification<sup>4</sup>(OAS), which provides a language-agnostic interface to HTTP (Hypertext Transfer Protocol) APIs that lets humans and computers discover and comprehend the service's capabilities without source code, documentation, or network traffic analysis, currently in version 3.1.0.

Leonard Richardson<sup>5</sup> published a model in 2018 that breaks down the components of REST into three stages: the introduction of resources, the use of HTTP verbs, and the implementation of hypermedia controls. This approach is known as the Richardson Maturity Model (RMM), and it categorizes the design of a web application programming interface (API) into four levels. The higher the maturity level of the API, the more compliant the API is in terms of correctly employing the semantics of the HTTP verbs.

---

<sup>1</sup>Postman is an API platform for building and using APIs: <https://www.postman.com/state-of-api/api-global-growth/>

<sup>2</sup>REST API Guidelines: Uniform Interface; Client-Server; Stateless; Cacheable; Layered System; Code on demand.

<sup>3</sup><https://restfulapi.net>

<sup>4</sup><https://spec.openapis.org/oas/v3.1.0>

<sup>5</sup><https://www.crummy.com/>

One of the biggest challenges in web API usage is validating that what has been specified at the API level, in our case, the OpenAPI schema, corresponds to the actual working and correct of the application. However, APIs are often prone to issues like idempotence violations, access control problems, and improper handling of resource dependencies, which can introduce significant risks.

In the context of REST APIs, other types of validation must be taken into consideration, such as security, performance, idempotence, load testing, functional, and properties that are not specified at the code level, such as the correct use of HTTP verbs, ensuring the correct sequences of requests are followed (e.g., `CREATE` before `DELETE`), or how the application deals with method restrictions (e.g., methods not allowed). Traditional API testing methods may not be sufficient to detect these kinds of problems.

In recent years, tools have emerged that support the creation of automated testing based on the OpenAPI specification, which generates random requests to maximize the number of possible combinations of tests to identify flaws and security vulnerabilities.

Even though these tools have proved to be very useful in generating automated tests, there are still many unexplored areas where we can work; with this in mind, our purpose is to extend an automated testing tool such as RESTler [2] to support validation of idempotence, dependencies and side-effects between requests specified at OAS schema level, and therefore support the testing of security properties.

## 1.1 Problem Definition

Creating automated tests to validate APIs has become more and more critical. Tools and methods have emerged recently to help deal with this exponential growth. But there's still plenty of opportunity to work on, such as testing properties that aren't usually defined at the API specification level or specify relationships between requests. Despite the critical importance of APIs in software systems, there are several key challenges that we need to deal with when validating API behaviour, such as:

- **Idempotence Violations:** Operations like `PUT`, `DELETE` should be idempotent, meaning multiple identical requests should result in the same outcome. However, without careful testing, these operations may lead to inconsistent results.
- **Resource Dependency:** Many APIs have interdependent resources, e.g., a `DELETE` should follow a `POST`, and incorrect sequence handling can cause errors.
- **Method Misuse:** Endpoints are intended to support only certain HTTP methods, e.g., `GET`, `POST`, and using incorrect methods may lead to unexpected or undocumented behaviours.

## 1.2 Objectives

The main objectives of this study are to:

- **Identify** properties in RESTful API for testing.
- **Propose** an extension for OpenAPI to describe relationships between operations.
- **Extend** a tool that performs stateful tests to validate the identified properties.
- **Run** the tool with the extension in an API with a well-structured schema to validate the proposed method.

To accomplish these goals and provide direction for our research endeavours, the following research questions were formulated:

**RQ1** – How effective an automated testing tool like RESTler is in identifying violations of RESTful principles like idempotence, sequencing, operation dependencies and ensuring API security, especially concerning method restrictions (*e.g., methods not allowed*)?

**RQ2** – What is the impact of adding an extension to specify related operations in OpenAPI on API testing? Can this affect automated testing strategies?

## 1.3 Contribution

This study aims to make the following contributions:

- **Literature Review** – This comprehensive review categorises and analyses works on automated testing tools connected to this topic. The goal is to provide a methodology to establish a validation mechanism for commonly untested properties.
- **Propose an OpenAPI Extension** – To make it easier for humans and machines to recognise the relationship between operations, in this case, automated test generation tools, an extension to OpenAPI is proposed.
- **Extension Validation** – Usage of the proposed extensions to OpenAPI and the testing tool to validate their effectiveness.

## 1.4 Dissertation Structure

In addition to the introduction, this dissertation contains 6 more chapters. In Chapter 2, we catch up on essential and relevant background topics to help us understand the following sections. In Chapter 3, the state of the art is described, and related works are presented. In Chapter 4, we discuss how RESTler works and why it is the best tool for the scope of our research. In Chapter 5, the design and implementation of the active checkers will be detailed. The evaluation and discussions are presented in Chapter 6. In the final Chapter 7, we will present our conclusions based on the conducted research.

## Chapter 2

# Background

To ensure that the sections that follow are clear and understandable, this chapter provides necessary background information. Roy Fielding formalized the principles of REST and further detailed them in specifications such as RFC 9110, offering a standardized framework for API design. The OpenAPI Specification defines a standard, language-agnostic interface for describing RESTful services and typically documents these APIs. The next section delves into the significance of fuzzing testing in API testing, while the final section presents the Richardson Maturity Model (RMM). This model provides a framework for evaluating an API's maturity by considering its compliance with RESTful principles, appropriate use of HTTP methods, status codes, and hypermedia controls, among other factors.

### 2.1 REST and RFC 9110

As described in previous sections, REST was introduced by Fielding in 2000 as an architectural style that defines a set of guidelines for designing and implementing APIs; since then, it has been used by most web applications. A REST API can be implemented in different programming languages; this gives developers the freedom to implement a REST API in various ways without being restricted to specific design patterns.

In a REST API, any information can be transferred as a resource, such as files, images, text, etc. This information, or representation of the resources, is usually transferred using the JSON (Javascript Object Notation) format, other data formats used in REST APIs are XML (eXtensible Markup Language), HTML (Hypertext Markup Language) and YAML (YAML Ain't Markup Language).

The actions taken on the resources are carried out using HTTP verbs as defined on RFC 9110 [5], such as GET, POST, PUT, PATCH and DELETE. These verbs express CRUD (Create, Read, Update and Delete) operations. For example, bank account information can be retrieved by invoking the URL for an account (*\$baseUrl/accounts/01500447*), where 01500447 is the number identifier of the intended account.

Defined by RFC 9110, the HTTP response status codes are split into 5 clusters: informational responses (100–199), successful responses (200–299), redirection messages (300–399), client error responses (400–499), server error responses (500–599), each of these groups indicates whether a request made via HTTP has been completed and what the status of this request. Figure 2.1 shows the most commonly used HTTP status codes.

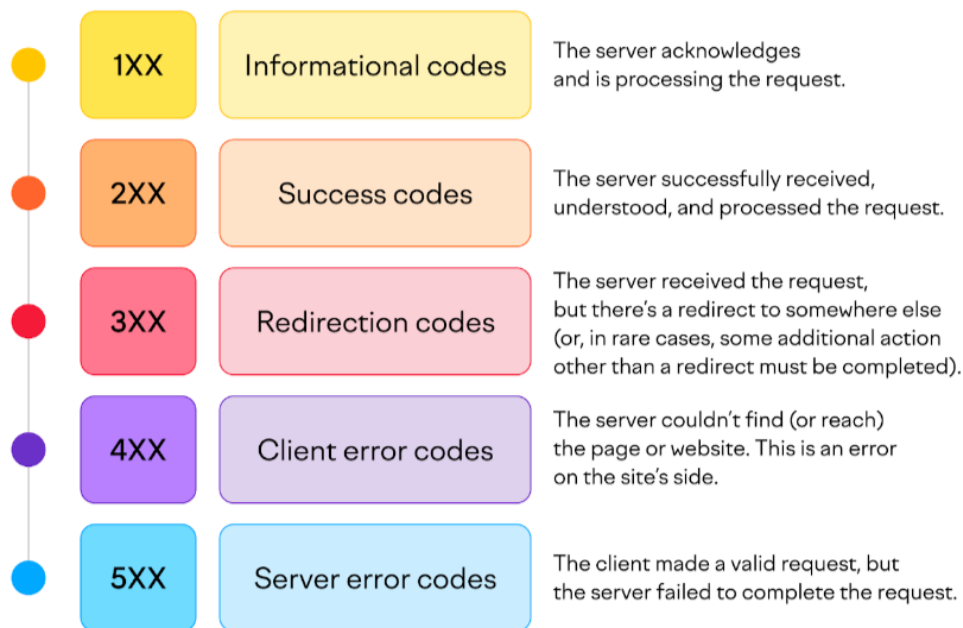


Figure 2.1: HTTP Commons Status Code<sup>1</sup>

In the automated tests, the HTTP status code defined in the OAS plays an important role, as it is used to check whether the result received during test execution corresponds to the result specified so the test generation tools can define valid and invalid tests.

## 2.2 OpenAPI Specification

Without the need for additional documentation, network traffic inspection, or access to source code, the OpenAPI Specification (OAS) [14] provides a standard, programming language-neutral interface description for HTTP APIs that enables both computers (m2m) and humans to learn about and comprehend a service's capabilities. A minimal implementation logic is required for a consumer to comprehend and engage with a remote service correctly specified using OpenAPI. The OpenAPI Specification eliminates guessing when calling a service, much like interface specifications have done for programming at a lower level.

The OAS defined an OpenAPI document as a self-contained or composite resource which defines or describes an API or elements of an API. The OpenAPI document **MUST** contain at least one paths field, a components field or a webhooks field. An OpenAPI document uses and

<sup>1</sup>Source: <https://www.semrush.com/blog/http-status-codes/>

conforms to the OpenAPI Specification, the Listing 1 shows a sample of an OAS document in JSON format, where we can see on line 8 the endpoint and on line 9 the operation (HTTP verb POST) available by this operation, the field `requestBody` which indicates which properties are expected when making the request, and starting on line 33 we have the field representing the responses that the application can send.

```
1 {
2   "openapi": "3.0.0",
3   "info": {
4     "title": "Auth Specification",
5     "version": "1.0.0"
6   },
7   "paths": {
8     "/auth/register": {
9       "post": {
10        "security": [],
11        "tags": [
12          "Authentication"
13        ],
14        "requestBody": {
15          "content": {
16            "application/json": {
17              "schema": {
18                "type": "object",
19                "properties": {
20                  "email": {
21                    "type": "string"
22                  },
23                  "password": {
24                    "type": "string"
25                  }
26                }
27              }
28            }
29          }
30        },
31        "required": true
32      },
33      "responses": {
34        "201": {...},
35        "400": {...}
36      }
37    }
38  }
39 }
```

Listing 1: OpenAPI schema for `/auth/register`

Among the various properties available in OAS, it is worth highlighting the Links, through the Link Object<sup>2</sup>, the standard OpenAPI specification enables the definition of connection between resources, e.g., when using Links we can specify that a value that is returned by one operation can be used by another, the Figure 2.2 shows an example of this specification in OAS. For instance the operation of creating a blogpost `POST /api/blog/posts` returns a **postId**, we can specify via links that this **postId** can be used later as a parameter to read `GET /api/blog/post/12`, UPDATE or DELETE a blogpost.

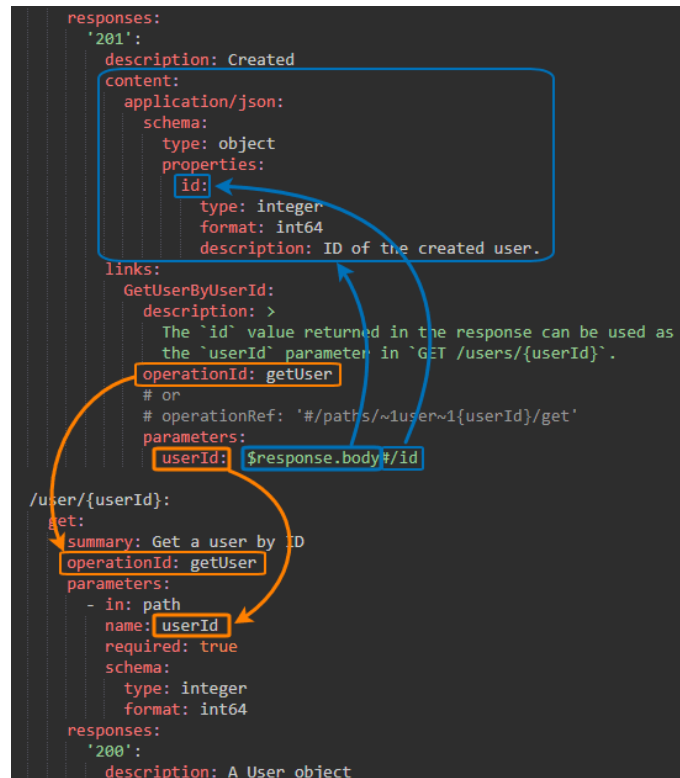


Figure 2.2: Links in Swagger Spec<sup>3</sup>

## 2.3 Fuzzing Testing

As the complexity of APIs increases, ensuring their correctness, security, and reliability becomes crucial. One of the emerging methods to address these challenges is fuzz testing. This automated technique generates and sends inputs to APIs to detect unexpected behaviours or vulnerabilities. Patrice Godefroid<sup>4</sup>, in the article [7] defines fuzzing in the following way:

Fuzzing, or Fuzz Testing, is the process of finding security vulnerabilities in input-parsing code by repeatedly testing the parser with modified or fuzzed inputs.

<sup>2</sup><https://spec.openapis.org/oas/v3.1.0#link-object>

<sup>3</sup>Source: <https://static1.smartbear.co/swagger/media/images/links.png/>

<sup>4</sup><https://patricegodefroid.github.io/>

This approach has proved to be very effective in detecting security vulnerabilities in REST APIs, and has become standard within the development cycle of some software companies, such as Microsoft, which has adopted this approach as a standard for validating its products. Fuzz testing can be categorised as follows:

- **Blackbox Fuzzing** – This is the simplest way to perform fuzzing. This approach generates random values to detect vulnerabilities or errors starting with a valid input.
- **Grammar-Based Fuzzing** – This approach uses a structured input, such as XML, JSON or YAML files. For REST API, the use of the OAS schema plays a fundamental role, as it can provide the tool with information such as the type of data supported by a given field, the minimum and maximum acceptable size, and with this, the tool can generate randomised tests to validate these definitions.

Like the blackbox fuzzing, the quality of the input matters because if we don't have a well-documented schema, the values generated will make noise, and possibly invalid tests will be created.

## 2.4 Maturity Models in Detail

As briefly mentioned earlier, the RMM was developed by Leonard Richardson<sup>5</sup> and presented during the QCon in 2008<sup>6</sup>. This model categorises an API into levels, and the higher the level, the more this API complies with what is defined as REST [6], as can be seen in Figure 2.3. The

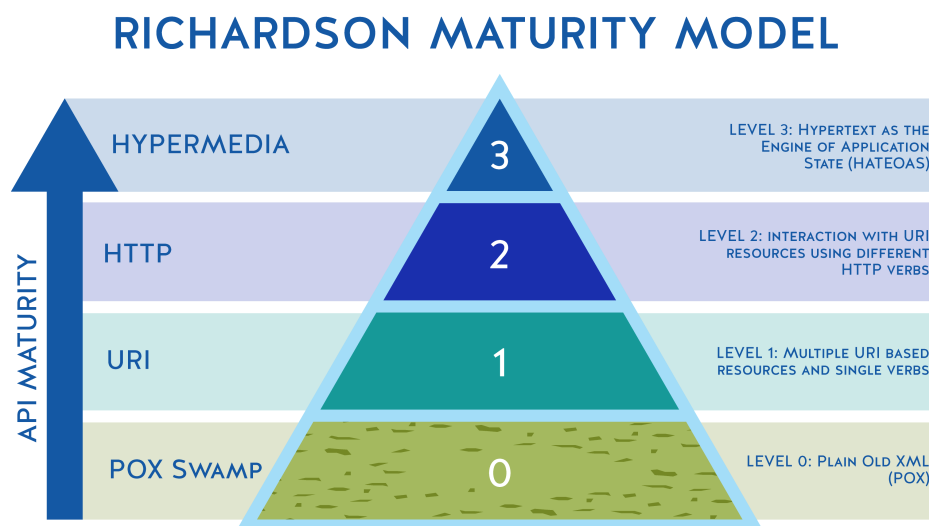


Figure 2.3: Richardson Maturity model<sup>7</sup>

<sup>5</sup><https://www.crummy.com/>

<sup>6</sup><https://www.crummy.com/writing/speaking/2008-QCon/act3.html>

<sup>7</sup>Source: <https://devopedia.org/richardson-maturity-model>

following subsections will be dedicated to looking at each level in more detail to understand their implications.

### 2.4.1 Level 0 - Swamp of POX

This level illustrates an API in its most basic design; at this point, it only uses HTTP as a transport mechanism, having only one single URI endpoint and typically making use of the HTTP `POST` method for all operations, other HTTP verbs like `PUT`, `UPDATE` are ignored in this level.

In this step, I want to create a post entry in my blog system; for this, the application provides a single endpoint `$baseUrl/blogService` without any direct reference in terms of URI to what is intended to be done in the application, what kind of resource will be manipulated. On Listing 2 shows a request fragment where the desired operation `createPost` is specified in the request payload, on Listing 3 shows the response of this request.

```
POST $baseUrl/blogService HTTP/1.1
{
  "action": "createPost",
  "body": "my first blog"
}
```

Listing 2: RMM0 Request

```
HTTP/1.1 200 OK
{
  "status": "success",
  "postEntity": {
    "postId": 99,
    "body": "my first blog"
  }
}
```

Listing 3: RMM0 Response

APIs provide security risks at this level due to their use of a single endpoint and lack of distinction between types of resource kinds and activities, in a context where API access is not via an API Gateway or other mechanism that can control and segregate access to the APIs.

Excessive endpoint exposure can hinder the implementation of precise security measures. However, some mitigation measures can be implemented, such as validating and sanitizing inputs and outputs and implementing strong authentication and authorization mechanisms.

An API is classified as level 0 when it has the following characteristics: having a single endpoint, meaning that it has only one URI for the whole API; there is also a single HTTP method that is used, which is HTTP `POST`; and lastly, the action to be performed is contained in the request body.

### 2.4.2 Level 1 - Resources

Although Level 1 introduces the idea of resources, it does not yet entirely use HTTP methods in a RESTful manner. Several endpoints are used for various resources rather than just one. Nevertheless, only one HTTP method, usually `POST`, may be used for activities on these resources, with the action being provided in the request payload.

Suppose I want to create a post entry and comment in my blog system. In that case, the application provides different endpoints `$baseUrl/blogService/posts` to manipulate post entries and `$baseUrl/blogService/comments` to manipulate comments entries. On Listing 4 shows a request fragment where the use of resources is already presented and the desired operations `createPost`, `getPost` and `createComment` is specified in the request payload, on Listing 5 show the response of this request.

Although adding unique resources makes things more organised, they might not fully leverage HTTP techniques, which could result in less predictable behaviour. Applying resource-level access controls and ensuring that every resource endpoint adheres to the least privilege principle can prevent some security risks associated with this implementation level.

In summary, these are some of the characteristics of level 1: different URIs for different resources (e.g., `$baseUrl/blogService/posts` and `$baseUrl/blogService/comments`); as level 0 only HTTP `POST` is used for all operations; and the action to be performed is specified in the request body, similar to Level 0 but scoped to specific resource endpoints.

### 2.4.3 Level 2 - HTTP Verbs

By employing several HTTP methods to carry out distinct activities on resources, Level 2 enhances Level 1. Due to its utilisation of the HTTP protocol features, this design represents a considerable step towards becoming more RESTful.

At this level, the manipulation of resources will be done through their own URI and using the correct HTTP method, for example, if I want to create a comment on a blog, I'll use HTTP `POST`, and I'll have an endpoint similar to this `$baseUrl/blogService/comments`, to retrieve a comment I would use a `GET $baseUrl/blogService/comments/{commentId}`, and so on. On Listing 6 shows a request fragment where the use of resources and HTTP methods is already presented, on Listing 7 shows the response of this request.

This level has its own security advantages, as using HTTP verbs correctly, APIs can make their interactions more secure and predictable. For instance, applying `GET` for read-only operations and `POST/PUT` for state changes helps to put relevant security controls in place. On the other hand, some dangers exist regarding the wrong use of HTTP functions, including the exposure of confidential information via `GET` requests.

Level 2 is a more RESTful level than level 1 as it begins to use the entire HTTP functionality, which gives it a more conventional method of interacting with entities, and it is characterised by multiple endpoints: different URIs for different resources; multiple HTTP Methods: `GET`, `POST`, `PUT`, `DELETE`, etc., for different operations; and URIs identify resources, and operations on them are indicated by the HTTP method used.

### 2.4.4 Level 3 - Hypermedia Controls (HATEOAS)

According to the Richardson Maturity Model, level 3 is the highest level of maturity. The API now includes hypermedia controls, sometimes known as HATEOAS, or Hypermedia As The Engine Of

```
POST $baseURL/blogService/posts HTTP/1.1 #To create a new Post entry
{
  "action": "createPost",
  "body": "my first blog"
}

POST $baseURL/blogService/posts HTTP/1.1 #To retrieve a Post entry
{
  "action": "getPost",
  "postId": "99"
}

POST $baseURL/blogService/comments HTTP/1.1 #To create a new Comment entry
{
  "action": "createComment",
  "postId": 99,
  "comment": "new comment"
}
```

Listing 4: RMM1 Request

```
HTTP/1.1 200 OK #createPost entry response
{
  "status": "success",
  "postEntity": {
    "id": "99",
    "body": "my first blog"
  }
}

HTTP/1.1 200 OK #getPost entry response
{
  "status": "success",
  "postEntity": {
    "id": "99",
    "body": "my first blog"
  }
}

HTTP/1.1 200 OK #createComment entry response
{
  "status": "success",
  "commentEntity": {
    "postId": 99,
    "commentId": 1234
    "comment": "new comment"
  }
}
```

Listing 5: RMM1 Response

Application State. This makes the API self-descriptive because the API responses contain links instructing the client on what to do next.

On Listing 8 shows a request fragment where the use of resources and HTTP methods is

```
POST $baseURL/blogService/posts
Content-Type: application/json
{
  "body": "my first blog"
}

PUT $baseURL/blogService/posts/99
Content-Type: application/json
{
  "postId": 99,
  "body": "my first blog update"
}

GET $baseURL/blogService/posts/99

DELETE $baseURL/blogService/posts/99
```

Listing 6: RMM2 Request

```
HTTP/1.1 200 OK
{
  "id": "99",
  "body": "my first blog"
}

HTTP/1.1 200 OK
{
  "id": "99",
  "body": "my first blog update"
}

HTTP/1.1 200 OK
{
  "id": "99",
  "body": "my first blog update"
}

HTTP/1.1 200 OK
{
  "message": "Blog post deleted"
}
```

Listing 7: RMM2 Response

already presented, on Listing 9 show the response of this request with links, where we can see what we can do next, for example after the `POST` has been executed, line 1 on Listing 8, the response received indicates what I can do, for example, open the post just created, line 6, or even delete the post, line 8.

This level improves the quality of security by offering a self-descriptive API that is easy to navigate. Level 3 includes all the characteristics of an API on level 2, plus hypermedia controls, where responses include links that guide the client on what actions can be performed next, making the API more discoverable and self-documenting.

```
1  POST $baseUrl/blogService/posts
2  Content-Type: application/json
3  {
4      "body": "my first blog"
5  }
6
7  PUT $baseUrl/blogService/posts/99
8  Content-Type: application/json
9  {
10     "postId": 99,
11     "body": "my first blog update"
12 }
13
14 DELETE $baseUrl/blogService/posts/99
```

Listing 8: RMM3 Request

```
1  HTTP/1.1 200 OK
2  {
3      "id": "99",
4      "body": "my first blog",
5      "_links" :{
6          "self":{"href": /$baseUrl/blogService/posts/99},
7          "update":{"href": /$baseUrl/blogService/posts/99, "method": "PUT"},
8          "delete":{"href": /$baseUrl/blogService/posts/99, "method": "DELETE"},
9          "comments":{"href": "/$baseUrl/blogService/posts/99/comments", "method": "POST"}
10     }
11 }
12
13 HTTP/1.1 200 OK
14 {
15     "id": "99",
16     "body": "my first blog update",
17     "_links" :{
18         "self":{"href": /$baseUrl/blogService/posts/99},
19         "update":{"href": /$baseUrl/blogService/posts/99, "method": "PUT"},
20         "delete":{"href": /$baseUrl/blogService/posts/99, "method": "DELETE"},
21         "comments":{"href": "/$baseUrl/blogService/posts/99/comments", "method": "GET"}
22     }
23 }
24
25 HTTP/1.1 200 OK
26 {
27     "message": "Blog post deleted",
28     "_links" :{
29         "create":{"href": "/$baseUrl/blogService/posts", "method": "POST"}
30     }
31 }
```

Listing 9: RMM3 Response

## Chapter 3

# Literature Review

This section is organised as follows: we present the research approach, database and search queries, and the criteria for choosing the publications. Afterwards, analysed articles presenting existing tools and approaches and how they address the problem of generating automated tests are described in the **Tools** section.

### 3.1 Research Approach

With the exponential growth in the use of webapis and the offer of cloud services such as Microsoft Azure Cloud, Amazon Web Services (AWS), Google Cloud Platform (GCP), Oracle Cloud Infrastructure (OCI), where services are available via API, there have also been many studies in the area of REST API testing. We began by analysing which articles address the security testing problem, how API specifications have supported the definition of dependencies between resources, what is supported by current testing tools, and how the OAS supports the specification of relations between requests.

#### 3.1.1 Database and Search Queries

The most recognised databases in the scientific world were used to obtain relevant and reliable scientific information, such as IEEE, ACM, Web of Science, Scopus, Springer and Inspec. A detailed table of publications is described in Section [A.1](#).

With an initial search in the IEEE database, using the query below and restricting the initial results with the following filters available from IEEE: articles published after 2018, conferences, journals and magazines, it was possible to obtain approximately 206 articles.

```
("Document Title":restful OR "Index Terms":restful OR "Document Title":rest OR "Index Terms":rest) AND ("Document Title":test* OR "Index Terms":test*) AND ("Full Text & Metadata": "black box" OR "Full Text & Metadata":fuzz* OR "Full Text & Metadata": "property-based test*" OR "Full Text & Metadata": "integration testing" OR "Full Text & Metadata": "system test*")
```

Figure 3.1: IEEE Command Query

### 3.1.2 Paper Selection Criteria

The selection of articles was based on a thorough analysis of titles and abstracts to ensure relevance to the research topic, followed by an evaluation of the study’s methodologies, results, and alignment with the objectives of this thesis to ensure a comprehensive and robust literature foundation. As inclusion criteria were used: articles written in English, published after 2018 and related to testing REST APIs. PhD and MSc theses, books and surveys were also included. The surveys were essential in helping to define the research criteria and as a source to collect a set of APIs to be used in our validations.

With the results obtained from the IEEE database, out of a total of 206 articles, 32 were selected for a second analysis, as they had some relation to our topic, and of these 32 articles, around eight were selected to be analysed in more detail, in a refinement that was based on: relevance to research topic, publication date, scope of application, (e.g., case studies, frameworks, etc), specifically on the implementation of REST APIs and their behavior under certain conditions, as this aligns with the practical aspects of the API testing methods developed in this thesis.

## 3.2 Tools

As mentioned above, many automated testing tools for REST API have emerged in recent years. Theses and publications assessing these tools have been published to confirm their efficacy in issue detection. A selection of the tools that were highlighted in the articles will be discussed in the following subsections.

### 3.2.1 RESTler

The first automatic stateful REST API fuzzing tool, RESTler [2], was introduced in 2019 by Microsoft researchers. RESTler employs protocol fuzzing techniques to stress-test APIs by injecting malformed or unexpected inputs, ensuring that applications can handle unforeseen situations. RESTler currently relies on an OAS to represent a service’s input space and automatically learns how to prune invalid request sequences by analysing service responses. Figure 3.2 shows the high-level architecture of RESTler. RESTler operates in the following four main modes in sequential

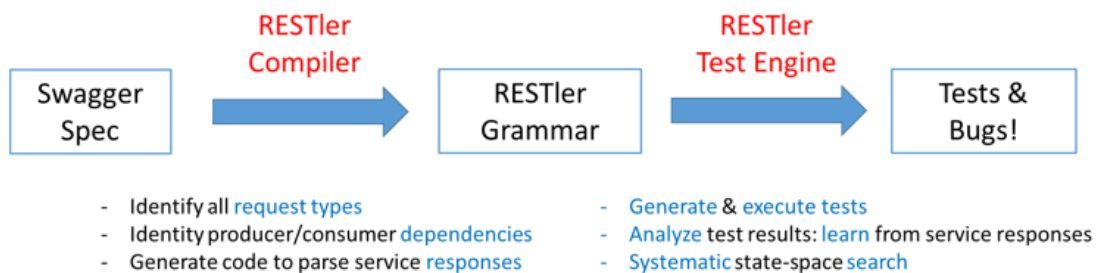


Figure 3.2: RESTler Architecture

order:

- **Compile** Generates a RESTler grammar from OpenAPI specification in JSON or YAML format.
- **Test** Quick execution of all API endpoints from the compiled RESTler grammar.
- **Fuzz-lean** This pattern groups those relations where the follow-up outputs should be subsets (or strict subsets) of the source output.
- **Fuzz:** Executes once every API endpoint from the compiled RESTler grammar with the default set of checkers.

In 2020, Microsoft researchers, together with a researcher from Columbia University, published a paper entitled Checking Security Properties of Cloud Service REST APIs [1] at the 13th International Conference on Software Testing, Validation and Verification (ICST), introduced four new security rules, in which they extended RESTler [2] to be able to capture specific properties of REST APIs and services.

These rules are known as active checkers, *unlike passive checkers that only observe responses without interfering, active checkers interact with the API by crafting and sending sequences of API requests and suggests new tests to assert that specific rules are not violated* [1]. These rules are described as follows:

- **Use-after-free rule** – A resource that has been deleted must no longer be accessible. For example, if a user account is deleted using the operation **DELETE** `$baseURL/api/user/123` and returning a status code 2xx, all subsequent requests should fail or return a status code 404 or 204.
- **Resource-leak rule** – A resource that was not created successfully, response different 2xx, must not be accessible and must not “leak” any side-effect in the backend service state. When trying to create a user account with operation **POST** `$baseURL/api/user/`, it failed, sending a request that uses the resource that failed during creation should return an error, response different 2xx, otherwise a bug exists in the application.
- **Resource-hierarchy rule** – A child resource of a parent resource must not be accessible from another parent resource. For example, a comment that has been created on the blog with id **123** `$baseURL/api/blog/123/comment/321` this comment cannot be accessed within the post with id **124**, `$baseURL/api/blog/124/comment/321`.
- **User-namespace rule** – A resource created in a user namespace must not be accessible from another user namespace. For instance, a resource created using a specific token cannot be accessible in other requests using a different token.

To validate the proposed method, the authors have used cloud services currently in production, available through Azure and Office365, and were able to detect bugs in these services. Compared

to other articles, the implementation process is very detailed, and it is possible to reproduce the tests using another set of APIs.

RESTler was implemented using Python, and the code was split into the parser and compiler module, the core fuzzing runtime module, and the garbage collector (GC) module. The usage is through CLI, where the primary input is the OAS, which deduces dependencies between request types in producer-consumer relationships based on the OAS and dynamically adapts to the service's behaviour by learning from previous responses. This tool will be detailed in chapter 4.

### 3.2.2 Schemathesis

Motivated to find a way to decrease the effort related to writing tests and a solution for identifying defects in the applications developed by Dmitry Dygalo<sup>1</sup> team's, and with the need to have a testing tool that could seamlessly assess applications written in any programming language, demand minimal maintenance, and offer user-friendly functionality, have decided to develop Schemathesis<sup>2</sup>.

Schemathesis is a tool for automated black box or white box randomised testing of web APIs, deriving structure and semantics-aware fuzzes from OpenAPI or GraphQL schemas. In this article [8], the authors give a comprehensive overview of the use of these tools, where they address the following points:

- describe Schemathesis, a property-based testing library to automatically derive customisable web API fuzzers from OpenAPI or GraphQL schemas;
- demonstrates that Schemathesis discovers more defects and handles more real schemas than previous tools and
- provides a large evaluation suite of real web services, schemas, and fuzzers as reusable containers for use in future research.

### 3.2.3 RESTest

In 2021, at the International Symposium on Software Testing and Analysis (ISSTA), the RESTest [12] was the only tool supporting constraint-based testing of RESTful APIs. Like other tools, RESTest receives the OAS as an input parameter; Figure 3.3 shows the workflow proposed to run this tool.

Brief description of the RESTest workflow [12]:

1. *Test model generation.* RESTest follows a model-based testing approach. Two models are used: the system model (i.e., the API specification) and the so-called test model, consisting of a configuration file in YAML notation.

---

<sup>1</sup>Part of Testing Platform team at Kiwi.com

<sup>2</sup><https://docs.schemathesis.io/>

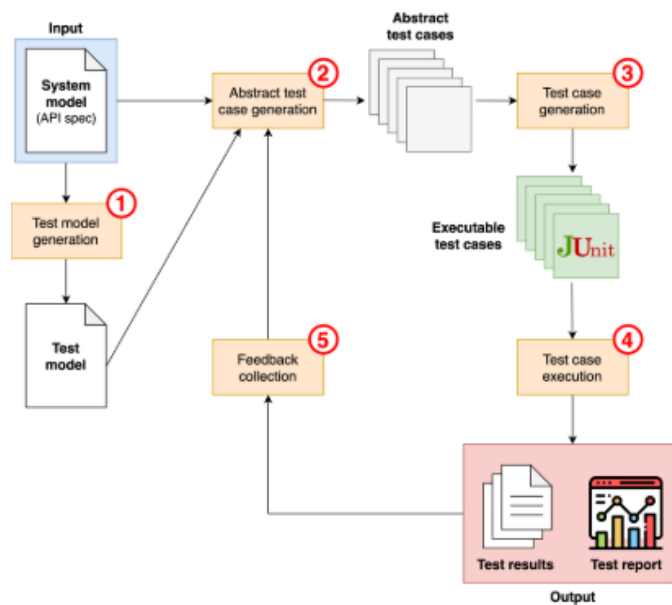


Figure 3.3: RESTest workflow

2. *Abstract test case generation.* Test cases are derived from the system and test models using one or more testing techniques.
3. *Test case generation.* Abstract test cases are instantiated into executable test cases using specific testing frameworks and libraries such as REST Assured<sup>3</sup>.
4. *Test case execution.* Test cases are optionally executed, and the test results are exported to a machine-readable format and reported to the user.
5. *Feedback collection.* Test generators can react to the test outputs to create more sophisticated test cases, for example, applying search-based techniques to maximise the API coverage.

### 3.2.4 GADOLINIUM

The article [3] presents a different and exciting tool, where the authors introduce a framework to monitor the non-functional properties of webapis. The OAS does not define any structure for specifying Non-Functional Properties (NFP) such as performance and availability; with this in mind, the authors present a framework called GADOLINIUM<sup>4</sup>. Like other tools, the OAS is used as input to conduct tests to test the NFPs of the operations. The current solution supports two types of NFP: performance (*latency*) and availability (*uptime*).

Performance is measured by taking into account the time it takes to make a request and the time it takes to receive a response back; availability is measured according to the time the API is live to accept requests. As a result of these tests, the tool makes it possible to enrich OpenAPI

<sup>3</sup><http://rest-assured.io>

<sup>4</sup><http://hdl.handle.net/20.500.12004/1/A/GADOLINIUM/001>

description using the standard OAS extension mechanism with these values, which can then be used as a reference for consumers of these APIs.

Even while this study does not explicitly address the topic of security, it does introduce NFPs as target characteristics to be examined and indicates how OAS can be extended. Additionally, it demonstrates how the same tool can be extended to carry out security NFPs, despite the fact that it initially only covers the NFPs outlined above.

Since 2017, the OWASP top 10 Web Application Security Risks<sup>5</sup> lists sensitive data exposure as a risk. In this article [4], the authors propose an approach for carrying out tests to discover security problems related to the exposure of sensitive data fields and insecure API calls that access sensitive data; these sensitive and non-sensitive data fields are exposed through the same API calls.

The start point of this study was to analyse the OpenAPI documentation to figure out the interdependency relationship between requests and data fields that can carry useful information. The following implementation aspects can be inferred from this: which data are likely to be sensitive, which API call needs more protection, and which data fields are more exposed. It is possible to generate information that can support improving the API design. Figure 3.4 shows the semi-automated workflow of the security analyses and the interaction needed from the developer side. With this, the developer can label a shortlist of fields to indicate sensitive or non-sensitive data without analysing hundreds of fields in the schema.

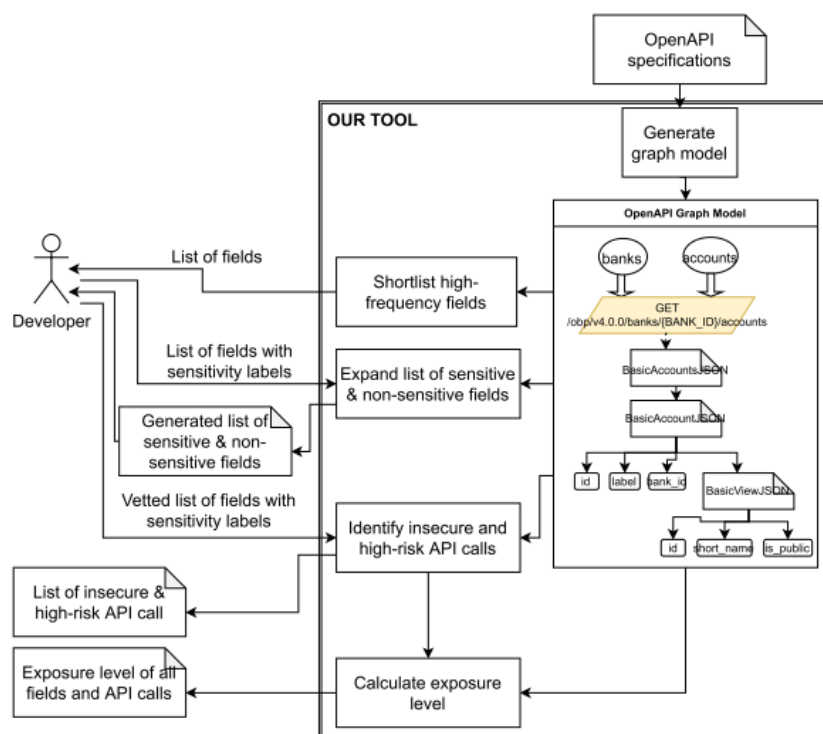


Figure 3.4: The workflow of proposed security analyses<sup>6</sup>

<sup>5</sup><https://owasp.org/www-project-top-ten/>

The identification of *sensitive and non-sensitive data* reduces developers' effort to carry out this identification manually because the set of data fields that must be manually identified is reduced. The authors also identified that sensitive fields tend to be grouped because they are usually part of the same object and because they are used in several API calls.

The authors have used the Open Bank Project's API specifications to validate the proposed approach. The Open Bank Project (OBP) is a global standard and open-source API solution for open banking used by more than 100 financial institutions and regulators worldwide. The study helped identify 31 sensitive data fields not identified in manual analysis, 29 API calls with insufficient authentication controls and 34 high-risk API calls that provided access to sensitive and insensitive data without extra security mechanisms to protect these fields.

Identifying dependencies between APIs is not a simple task. To alleviate the problem, current tools build these dependencies using the breadth-first search<sup>7</sup> (BFS) approach to create valid test cases. However, depending on the number of APIs in a schema, this approach has proved inefficient in terms of performance since generating test cases can take too long, given the need to generate dependencies between all the nodes in an API. Thus, the main difference between this approach and traditional approaches using the BFS is that instead of modelling API dependencies based on their required parameters and response values, they propose to derive API hierarchies by considering the endpoint of each API.

### 3.2.5 foREST

In the article [10], the authors propose an approach to identifying dependencies, such as parameters, fields, or resources, between APIs using a structured tree, consequently enhancing the dependency complexity, shifting it from quadratic (when traversing a graph) to linear (when traversing a tree) concerning the number of APIs, an API is uniquely defined with an endpoint and a method (GET, POST, PUT, or DELETE).

The approach proposed consists of two main stages: a preparation stage that is composed of an API Tree Construction and Resource Annotation, and another stage called the Fuzzing Stage that is formed by Test Case Generation, on this stage is created test to validate the order of HTTP Methods identified in each node, Test Case Execution, Response Analysis and a Resource Pool that is used to store the dependent resource, the Figure 3.5 shows the framework proposed.

The first stage organises the APIs into trees, and the fuzzing stage generates the test cases by navigating through the API trees and executing each of them using the designated REST service.

This approach detected two sets of errors in the GitLab and WordPress APIs: Invalid Parameters and Logical Errors. These errors can be sub-categorised as follows:

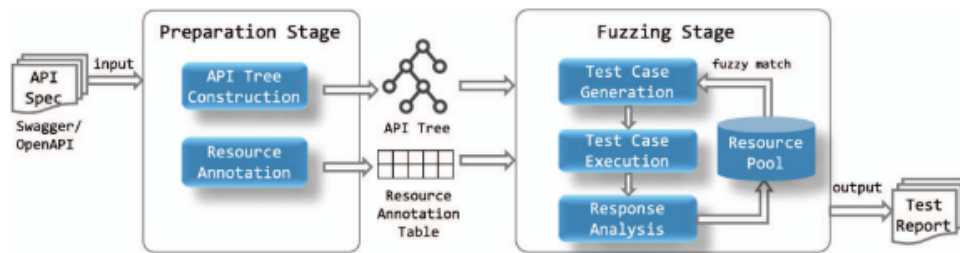
- Invalid Parameter

---

<sup>6</sup>Source: <https://ieeexplore.ieee.org/mediastore/IEEE/content/media/9652566/9652628/9652639/cheh2-317000a015-large.gif>

<sup>7</sup><https://brilliant.org/wiki/breadth-first-search-bfs/>

<sup>8</sup>Source: <https://ieeexplore.ieee.org/mediastore/IEEE/content/media/10299935/10299936/10301255/lin4-p11-lin-small.gif>

Figure 3.5: Framework of foREST<sup>8</sup>

- UTF-8
- special Char
- long str with special char
- enum type with bad value
- Logical Error
  - use after delete
  - double delete
  - duplicated id
  - reference loop
  - false logic

To exercise logical errors, the authors used GitLab, where they created a project by importing another project based on a specific value. To import a project, it is necessary to specify the url of the project to be imported, and GitLab validates whether the url protocol is http:// or https://, not whether the URL itself is valid. As a result, a request was made using the wrong URL, e.g. http://fakeurl, and GitLab returned a project ID, but if this ID were used in the next run, however, if further operation with the new project ID, GitLab would return 500. Figure 3.6 shows a bug of GitLab labelled as false logic.

```

1 Step 1: Create a project with an invalid
  "import_url"
2   Request: "POST /projects?name=a"
3     message: {import_url: http://badurl}
4   Response: code: 201
5     message: {id:1}
6 Step 2: Create a branch within this project
7   Request: "POST /projects/1/repository
8             /branches?branch=main&ref=main"
9   Response: code: 500
10    message: {"Internal Server Error"}

```

Figure 3.6: Sample bug detected by foREST<sup>9</sup>

In this same line of research to identifying dependencies between APIs, the author A. Martin-Lopez, in the articles [13, 11], addresses a problem related to *inter-parameter dependencies*. Sometimes, APIs restrict how two or more input parameters must be combined to form a valid call. For example, the YouTube Data API states that when using the parameter `videoDefinition` (e.g. to search videos in high definition), the type parameter must be set to 'video'; otherwise, an HTTP 400 code (bad request) is returned [11]. This type of restriction is not supported by the current OAS, which offers no way of specifying these restrictions, leaving it up to the developers to document these restrictions in natural language, which can lead to inconsistencies or not having the results expected by those who use these APIs.

After analysing a set of 40 RESTful APIs, the author concluded that it is possible to find some dependency between the input parameters in 85% of these APIs. These dependencies between parameters have been classified into seven types: *Requires*, *Or*, *OnlyOne*, *AllOrNone*, *ZeroOrOne*, *Arithmetic/Relational* and *Complex*.

To collect the information about dependencies, the author used the information provided in the documentation of each APIs analysed, which made it possible to identify the number of parameters involved in each constraint. In addition to manual analysis, they also used a text analysis tool<sup>10</sup> to identify linguistic patterns used to describe each type of dependency.

Another interesting approach is presented by Sergio Segura, José A. Parejo, Javier Troya and Antonio Ruiz-Cortés, who introduce the metamorphic relation output patterns (MROP) concept. An MROP defines an abstract output relation typically identified in Web APIs, regardless of their application domain [15].

The MROP is divided into six patterns: Equivalence, Equality, Subset, Disjoint, Complete and Difference. These six patterns cover all possible logical relationships between sets of outputs that could result from transformations applied to inputs in metamorphic testing. This ensures that no significant type of output relationship is overlooked. The patterns represent a range of expected behaviours from strict (e.g., equality) to more relaxed or complex relationships (e.g., subset or complete). This variety helps cover a wide range of software functionalities. They are categorized this way:

- **Equivalence** This pattern represents those relations where the source and follow-up outputs are equivalent.
- **Equality** This pattern represents those relations where the source and follow-up outputs must contain the same items in order.
- **Subset** This pattern groups those relations where the follow-up outputs should be subsets (or strict subsets) of the source output.
- **Disjoint** This pattern represents those relations where the intersection among the source and follow-up outputs should be empty.

<sup>9</sup>Source:<https://ieeexplore.ieee.org/mediastore/IEEE/content/media/10299935/10299936/10301255/lin9-pl1-lin-small.gif>

<sup>10</sup><https://www.online-utility.org/text/analyzer.jsp>

- **Complete** This pattern includes those relations where the union of the follow-up outputs should contain the same items as the source output.
- **Difference** This pattern includes those metamorphic relations where the source output and the follow-up output should differ in a specific set of items.

This approach has proved useful because, as with the classic metamorphic testing (MT) approach, metamorphic relationships (MR) are identified/characterised based on how familiar the developer is with the API, i.e. what prior knowledge they need to be able to specify the metamorphic relationships more precisely.

## Chapter 4

# RESTler

Considering the tools obtained from the literature review, Section 3.2, and the specific criteria such as support for stateful testing, OpenAPI compatibility, extensibility, available documentation, and bug detection efficiency, RESTler was determined to be the most suitable tool for this project. Therefore, this chapter will provide a more comprehensive analysis of the features of RESTler.

### 4.1 Introduction

RESTler is a fuzzing tool that Microsoft researchers developed to help improve the security and reliability of Microsoft's cloud services. This tool covered a gap in existing solutions used to test REST API. Most of these existing tools perform testing using isolated requests, which means they have low coverage, use traffic captures, and retry requests with fuzzing, which depends on good traffic, or manual custom fuzzers, which have a high cost.

As described in the article [2], RESTler is considered the first stateful REST API fuzzing tool due to several unique features, such as stateful API testing; this means it recognizes that the sequence and order of API calls can affect the state of the application, uses the API specifications *typically provided in OAS format* to generate sequences of API requests, dynamically learns and adapts to the responses from the API during the fuzzing process, identifies and reports a variety of errors, such as security vulnerabilities, crashes, and unexpected behaviour, by analyzing the responses and status codes returned by the API.

To avoid redundant testing, RESTler applies some strategies. For example, if we have a sequence of requests (e.g., POST -> PUT -> GET) using a certain number of parameters and a particular executing ordering of them, we receive a 400 HTTP status code, then RESTler knows that this ordering with these parameters, the service already handles that with a valid error handling response, with this RESTler will not go to try that exact sequence again, this means that the RESTler is avoiding unnecessary executions, which reduces test execution time.

Based on OAS, RESTler can infer dependencies between requests using the producer-consumer concept, similar to what is made when the `links1` object is used on OAS, for example, if we have

---

<sup>1</sup><https://spec.openapis.org/oas/v3.1.0#link-object>

an operation `POST`, e.g., `$baseUrl/v1/user/` to create a user account and the return of this operation is `userId`, another operation `GET`, e.g., `$baseUrl/v1/user/{userId}`, which uses the `userId` as a parameter, the RESTler can infer that the `userId` used in the `GET` is the same `userId` that was created in the previous operation. When a relation cannot be automatically inferred, we can manually define these relationships in the `unresolved_dependencies.json` file, or annotations can be used to provide hints about dependencies and re-run RESTler to ensure that the coverage level has increased.

According to publications by RESTler's authors, RESTler has found the following bugs: API Specification, Input validation, Authentication, Resource management and Data leaks in cloud services. Many other types of bugs can be found by the RESTler using custom checkers, which can conduct tests on specific properties or areas we want to explore. Using this RESTler capability, we will develop checkers to test properties identified during the analysis phase. This point will be detailed in Chapter 5.

## 4.2 RESTler in Detail

RESTler has two main components, a **Compiler** and an **Engine**. Figure 4.1 shows these components, which can be run in four modes: Compiler, Test, Fuzz-lean and Fuzz. The following subsections will examine how these four modes work and what we can extract from each.



Figure 4.1: RESTler Main Components

The RESTler Compiler takes an OAS document and produces the fuzzing grammar and this fuzzing grammar has all the schema information about the requests, it has information about the dependencies, such as what resources created by one request are used by another request. The RESTler Test Engine takes the fuzzing grammar, possibly a custom fuzzing dictionary, and some other settings and runs tests.

### 4.2.1 RESTler Compiler

As previously mentioned, the RESTler Compiler, which is the first mode of four to be executed, takes OAS as input and generates a couple of configuration files, which can then be customized to possibly tune the coverage of our API or add additional values that we want fuzzed.

The Figure 4.2 shows an OAS of a blog post service, where we can `POST` or `GET` a blog post and then manipulate a blog post with these additional endpoints (`PUT`, `GET` and `DELETE`).

RESTler Compiler produces a grammar, as shown in Figure 4.2b; basically, the grammar says that once a blog post is created, we have an *id*, which is needed to manipulate it further, as we can see that directly from the paths in the OAS Figure 4.2a.

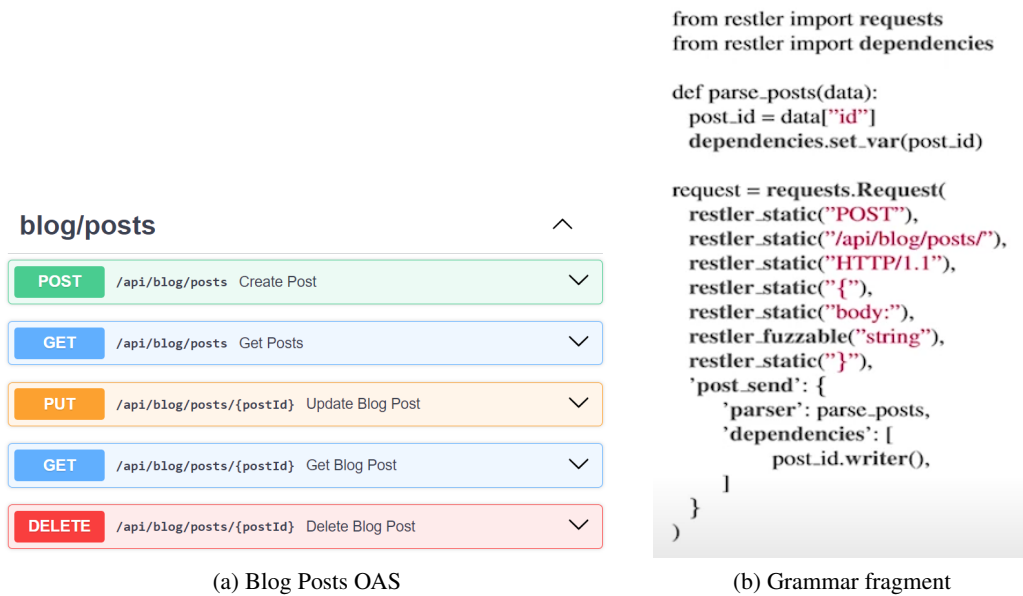


Figure 4.2: RESTler Compiler fragments

To run RESTler in compiler mode, the following command is used:

```

C:\restler-fuzzer\bin\rrestler\Restler.exe compile --api_spec
↪ C:\working-dir\demo-server-test\swagger.json

```

Listing 10: RESTler Compiler

In summary, RESTler, when executed in compiler mode, plays a fundamental role in preparing the API specification for the fuzzing process and achieving robust API testing. This mode produces several output files and artefacts, including the compile directory, the place that contains all files which will be used during the fuzzing mode; sequence files, the sequences respect the dependencies and constraints identified during the parsing phase; grammar files, including valid values for parameters, possible request bodies, etc.;

#### 4.2.2 RESTler Test

The RESTler Test is used to evaluate the API coverage, when this mode is executed, RESTler tries to run each operation at least once with success. This mode can be interpreted as *smoke testing*<sup>2</sup>, this means that this phase aims to ensure that the API is ready to move on to the next phase, in case of fail the request is marked as error and reflect this on the coverage level report.

<sup>2</sup><https://www.browserstack.com/guide/smoke-testing>

Based on the coverage level, RESTler offers a set of files, like a dictionary, config, etc., that we can adapt to increase the coverage level. Sometimes, the OAS has poor specifications, and RESTler cannot produce a connection between requests, which directly influences the coverage level. We must have a high level of coverage in this phase, as this will increase the likelihood of the RESTler finding bugs during the subsequent stages.

To run RESTler in test mode, the following command is used:

```
C:\restler-fuzzer\bin\restler\Restler.exe test --grammar_file
→ .\Compile\grammar.py --dictionary_file .\Compile\dict.json
→ --settings .\Compile\engine_settings.json --no_ssl
```

#### Listing 11: RESTler Test

When the command is executed, the RESTler Test engine kicks in, using the grammar generated in the previous phase, and then sending sample requests Listing 12 to be processed by the API. The results are displayed once the tests have been completed, as shown in Figure 4.3 where we can see the percentage coverage of successful requests and the total number of requests executed, Listing 13 show the response of the POST operation with the body of the message where we have the id generated and the body parameter with the information sent.

```
POST /api/blog/posts HTTP/1.1
Accept: application/json
Host: localhost:8888
Content-Type: application/json
Content-Length: 41
User-Agent: restler/9.2.4
x-restler-sequence-id: 317815b1-3dbe-4d6a-98c8-170b5d1d0aa5
{
  "body": "first blog"
}
```

#### Listing 12: RESTler Request Sample

```
Received: HTTP/1.1 201
Createddate: Thu, 25 Jul 2024 12:27:12 GMT
server: uvicorn
content-length: 28
content-type: application/json
{
  "id": 3,
  "body": "first blog"
}
```

#### Listing 13: RESTler Response Sample

```

Starting task Test...
Using python: 'python.exe' (Python 3.9.10)
Request coverage (successful / total): 6 / 6
Attempted requests: 6 / 6
No bugs were found.
Task Test succeeded.
Collecting logs...

```

Figure 4.3: RESTler Test output

### 4.2.3 RESTler Fuzz-Lean

This stage is the first time RESTler has been executed in a fuzzing mode. It runs in a lightweight, fuzzing systematic search to cover all APIs. Then, it runs the RESTler Checkers to fuzz them to find specific bugs, focusing on generating and executing a limited but representative set of API requests.

To run RESTler in fuzz-lean mode, the following command is used:

```

C:\restler-fuzzer\bin\restler\Restler.exe fuzz-lean --grammar_file
↪ .\Compile\grammar.py --dictionary_file .\Compile\dict.json
↪ --settings .\Compile\engine_settings.json --no_ssl

```

Listing 14: RESTler Fuzz-Lean

Once the fuzz-lean task is completed, the results are shown, as shown in Figure 4.4. Finding bugs, exercising the API with more requests, and running the checkers were possible during this stage. In this scenario, Figure 4.4b, a 500 status code error was found because the RESTler tried to access a resource that had already been deleted in previous requests, but if this resource were available after it had been deleted, the request would return a status code of 2xx and the checker would mark this request as a bug.

```

2024-07-26 11:29:11,676 - INFO - 127.0.0.1:56584 - "GET /api/blog/posts?page=10&per_page=10 HTTP/1.1" 200
2024-07-26 11:29:12,145 - INFO - 127.0.0.1:56584 - "POST /api/blog/posts HTTP/1.1" 201
2024-07-26 11:29:12,760 - INFO - 127.0.0.1:56584 - "GET /api/doc HTTP/1.1" 200
2024-07-26 11:29:13,450 - INFO - 127.0.0.1:56584 - "POST /api/blog/posts HTTP/1.1" 201
2024-07-26 11:29:13,513 - INFO - 127.0.0.1:56584 - "GET /api/blog/posts/4 HTTP/1.1" 200
2024-07-26 11:29:14,014 - INFO - 127.0.0.1:56584 - "POST /api/blog/posts HTTP/1.1" 201
2024-07-26 11:29:14,109 - INFO - 127.0.0.1:56584 - "PUT /api/blog/posts/5 HTTP/1.1" 204
2024-07-26 11:29:14,642 - INFO - 127.0.0.1:56584 - "POST /api/blog/posts HTTP/1.1" 201
2024-07-26 11:29:14,737 - INFO - 127.0.0.1:56584 - "DELETE /api/blog/posts/6 HTTP/1.1" 204
2024-07-26 11:29:15,113 - INFO - 127.0.0.1:56584 - "GET /api/blog/posts/6 HTTP/1.1" 500
ERROR: Exception in ASGI application

```

(a) Requests Fragment

```

Starting task FuzzLean...
Using python: 'python.exe' (Python 3.9.10)
Request coverage (successful / total): 6 / 6
Attempted requests: 6 / 6
Bugs were found!
Bug buckets:

UseAfterFreeChecker_500: 1
Task FuzzLean succeeded.
Collecting logs...

```

(b) Fuzz-Lean results

Figure 4.4: RESTler Fuzz-Lean output

In brief, during this stage, RESTler uses a structured approach that entails parsing specification files, request generation, dependency evaluation, and implementation of sophisticated fuzzing techniques RESTler employs to carry out its fuzzing activities. Various tests are created to engage with an API so thoroughly that one can study its actions and weaknesses. The outcomes from this fuzzing procedure, like logs, bug reports and coverage metrics, act as invaluable guides towards enhancing the strength and safety of an API.

#### 4.2.4 RESTler Fuzz

The RESTler Fuzz mode is the last mode to be executed, and according to the authors, it is not recommended to be executed in a production environment due to the load exerted on the system during testing. This mode is very similar to the previous one, fuzz-lean, but it focuses on generating a wide variety of requests, exploring different sequences, and applying various mutation strategies to input parameters, including boundary values, random data, large payloads, and invalid formats.

To run RESTler in fuzz mode, the following command is used:

```
C:\restler-fuzzer\bin\restler\Restler.exe fuzz --grammar_file
↪ .\Compile\grammar.py --dictionary_file .\Compile\dict.json
↪ --settings .\Compile\engine_settings.json --no_ssl
```

Listing 15: RESTler Fuzz

### 4.3 Extending RESTler

As previously mentioned, RESTler is a REST API fuzzing tool developed by Microsoft, the project is available on [github](https://github.com/microsoft/restler-fuzzer)<sup>3</sup> and can be cloned and altered to be used according to specific needs.

There are many ways to extend RESTler, including adding more fuzzing techniques, utilizing different request generation methods, introducing new logging mechanisms, and creating custom checkers for new kinds of validations and security checks that can be included in the fuzzing process.

CheckerBase is the base checker class from which all Python classes, marked as RESTler checkers, are derived, as shown in line 2 of the Listing 16, where we can see that the *CustomChecker* class takes as a parameter the CheckerBase class which is part of the core RESTler implementation. With this, the custom checker now has access to objects that are created by the RESTler during the compiler and test phases.

```
1 ...
2 class CustomChecker(CheckerBase):
3
```

---

<sup>3</sup><https://github.com/microsoft/restler-fuzzer>

```
4     def __init__(self, req_collection, fuzzing_requests):  
5         CheckerBase.__init__(self, req_collection, fuzzing_requests)  
6     ...
```

Listing 16: Checker Base Fragment

With this in mind, our approach will be to improve the logging mechanism to extract valuable information for analyzing the sequence of requests made by RESTler and implementing specific checkers. This implementation will be detailed in Chapter 5.

## Chapter 5

# Design and Implementation

The active checker design and implementation will be covered in detail in this chapter. Starting by explaining the case study of our approach, the following section details the design of the solution. Finally, it describes the proposed extension to OpenAPI specification, which aims to define relationships between operations and the implementation of the checkers in RESTler to finalise the chapter.

A high-level diagram of our suggested solution architecture is displayed in Figure 5.1 with fragments of our contributions located in the lower section.

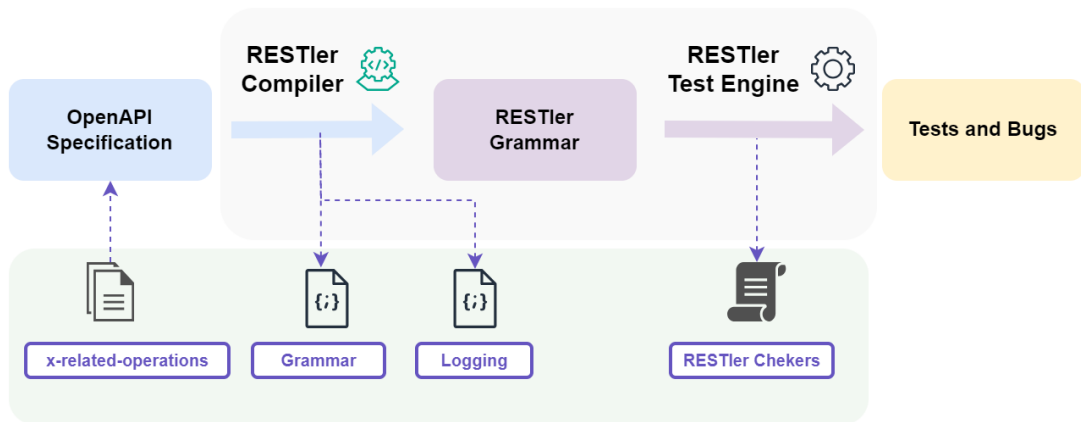


Figure 5.1: RESTler with Extended Components

As described above in Section 2.2, the standard of OpenAPI allows us to make connections between operations via the Link object, where the output of one operation can be used as input for another operation. However, this object only allows us to refer to the consumer and provider perspective without being able to specify details, such as what kind of dependency there is between the two operations or what effect it might have if one of the operations occurs. With this in mind, an extension **x-related-operations** to OpenAPI is proposed, which will be injected into the standard OAS schema.

The **grammar** and **logging** fragment, which is the standard RESTler file, will be enriched with information from the *x-related-operations*; finally, there are the **RESTler checkers**, which

will be implemented to extend the current capabilities of RESTler and which will be managed by the RESTler Test Engine.

The RESTler Checkers implemented targets a specific issue:

- **Sequence Checker** – verifies that operations are executed in the correct sequence.
- **Dependencies-Hierarchy Checker** – validates that the dependencies defined at schema level are executed in accordance with the implementation.
- **Idempotence Checker** – aims to ensure that operations behave consistently when repeated.
- **Method Not Allowed Checker** – detects incorrect handling of HTTP methods.

## 5.1 Current Case Study

For the proposed OpenAPI extension and custom checkers to be put into action, an application that simulates a Blog Post system based on REST architecture will be used, where we can create posts and comments on those posts. As this is a simple application, some errors were planted to validate that the proposed implementation would align with what is intended to be found in a real scenario. Figure 5.2 shows the context diagram of the Blog Post application.

This application<sup>1</sup> was built using FastAPI, a web framework to develop RESTful APIs in Python, and all tests were carried out on a computer with the following configurations: Windows 11 Pro, 16Gb RAM, and i5-8350U CPU @ 1.70GHz, 4 Cores, 8 Logical Processors.

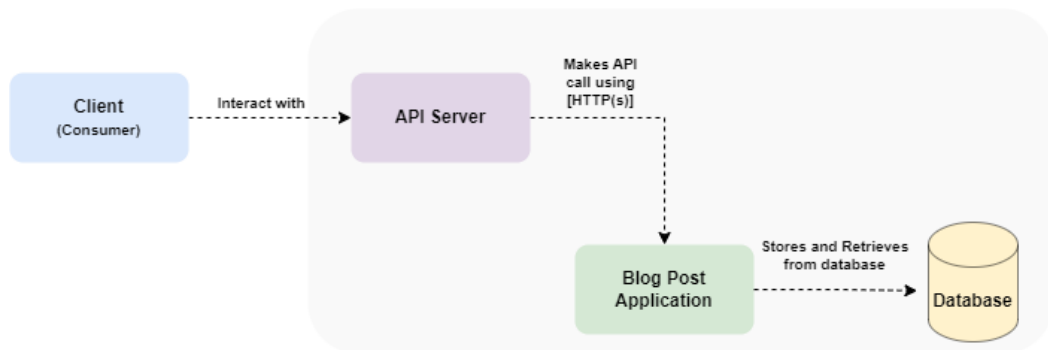


Figure 5.2: C4 Blog Post system context diagram

One of the principles advocated by REST proponents is that its state should be stateless; this means that every HTTP request from client to server must contain all the information necessary to understand and process it. The server does not keep any information in between requests, thus making each of them independent of previous ones. Such statelessness has its advantages, including simplification of the design process for servers, improving their scalability and enabling

<sup>1</sup>[https://github.com/microsoft/restler-fuzzer/tree/main/demo\\_server/demo\\_server](https://github.com/microsoft/restler-fuzzer/tree/main/demo_server/demo_server)

easier load balancing. However, this also means that tracking or managing state across multiple requests becomes difficult for servers.

As mentioned in Chapter 4, RESTler is a stateful testing tool, which means that it keeps a record of all the tests and sequences performed. It examines the API's specifications, generating requests based on dependencies between various API endpoints. Therefore, RESTler emulates interactive processes where, for instance, when creating a resource in one request, it would later be used in other requests, allowing for testing of stateful situations within a RESTful system context.

The Blog Post application used to validate our approach comprises nine endpoints: 5 related to Post operations, e.g., create, read, update and delete posts; three related to Comment operations, e.g., create, read and delete comments associated with posts; and one related to Audit operation.

There are eight endpoints:

- `POST /api/blog/posts` - Create Post
- `GET /api/blog/posts` - Get Post
- `PUT /api/blog/posts/{postId}` - Update Blog Post
- `GET /api/blog/posts/{postId}` - Get Blog Post
- `DELETE /api/blog/posts/{postId}` - Delete a Blog Post
- `POST /api/blog/posts/{postId}/comments` - Create Comment
- `GET /api/blog/posts/{postId}/comments/{commentId}` - Get Blog Post Comments
- `DELETE /api/blog/posts/{postId}/comments/{commentId}` - Delete Blog Post Comments
- `GET /api/blog/posts/{postId}/audit` - Get Blog Post Audits

Each resource is uniquely identifiable by a URI, and the OpenAPI schema was developed to allow it to express CRUD operations using the HTTP methods (GET, POST, PUT, and DELETE). The schema ensures a straightforward and RESTful design, *at least the RMM Level 2*, which employs standard HTTP capabilities to manage answers and provides the proper status codes.

## 5.2 Design

Since the information provided by the standard RESTler logs was insufficient to understand the API execution sequence and thus be able to define which areas we could explore, it was necessary to extend the tool to create a graph with this sequence. The Listing 17 shows a code snippet of this implementation.

For each request made to the server, the following information is collected: which action is being carried out represented by the `verb` (HTTP Method), line 17, the `endpoint`, line 18, that will be used to group the requests and the message sent in the `request_body` of the request, line 20. For the response, information is captured relating to the `status_code`, line 29, the message in the `response_body`, line 30, when applicable, and the `timestamp`, line 31, which will be useful for analysing the execution sequence in temporal mode. This graph<sup>2</sup> is represented graphically in Figure 5.3.

The RESTler was set to run in Test and Fuzz-lean modes during the design phase. The fuzz mode was not run because the information extracted in the two previous modes was sufficient for this stage to understand how the RESTler would execute the request sequences.

```

1  ....
2  def sendRecv(self, message, req_timeout_sec, reconnect=False):
3      """
4          Sends a specified request to the server and waits for a response
5      """
6      #Write in specific log information related to request and response
7      match = re.match(r'(GET|POST|PUT|PATCH|DELETE) (.*) HTTP/1.1',
8          ↪ message)
9      def _get_substring_between(string, start_char, end_char):
10         pattern = re.escape(start_char) + '(.*)' + re.escape(end_char)
11         match = re.search(pattern, string)
12         if match:
13             return match.group(1)
14         else:
15             return ""
16     if match:
17         request_summary = OrderedDict()
18         request_summary['verb'] = match.group(1)
19         request_summary['endpoint'] = match.group(2)
20         request_body = _get_substring_between(message, '{', '}')
21         request_summary['request_body'] = request_body
22         # Parse the URL into components
23         parsed_url = urllib.parse.urlparse(match.group(2))
24         # Extract query parameters
25         query_params = urllib.parse.parse_qs(parsed_url.query)
26         resource_id = parsed_url.path.split('/')[-1]
27
28         if resource_id is not None and query_params is None:
29             request_summary['resource_id'] = resource_id
30             request_summary['status_code'] = response._status_code
31             request_summary['response_body'] = response.body
32             request_summary['timestamp'] = iso_timestamp()

```

<sup>2</sup><https://jsoncrack.com/editor> generated this image-based in our request\_sequence.json file

```

32 write_incremental_requests(request_summary)
33 ...

```

Listing 17: Request-Response Sequence Graph

Each API endpoint is represented as a node in the generated graph, which also includes the sequence of requests made by each endpoint, in this case, grouped by resource and edge representation; for example, an edge from node A to node B indicates that a successful call to A provides the necessary input for B.

Here we have the URI `api/blog/posts/4`; for symbolic reasons, the slash (/) character has been replaced by an underscore (\_); in our request sequence fragment shown in Figure 5.3, we'll have two nodes: `_api_blog_posts_4` and `_api_blog_posts_5`.

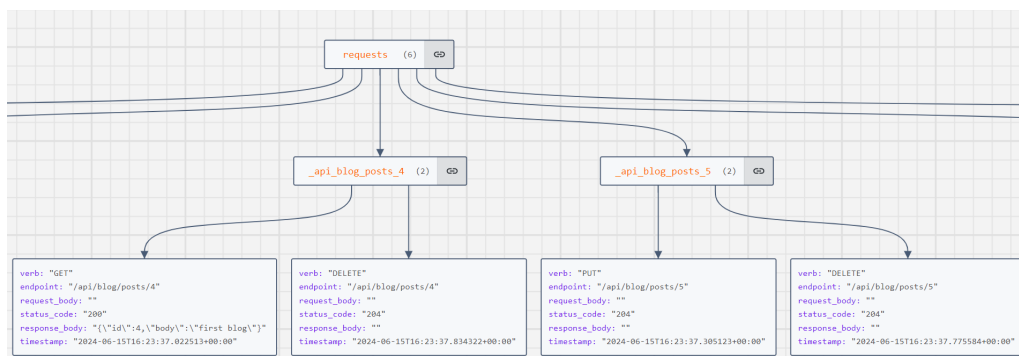


Figure 5.3: Request Sequence Fragment

By analysing this sequence of requests, it is evident that RESTler employs topological sorting<sup>3</sup> on the dependency graph to ascertain a legitimate request order. This sorting guarantees that all dependencies have been satisfied before submitting a request. Based on the sorted graph, RESTler constructs sequences of requests, starting with those that create, e.g. POST, resources and then those that depend, e.g. GET, PUT, DELETE, on these resources.

This behaviour of the RESTler can also be explained by its exploratory nature, where it always tries to create valid sequences of requests. However, this behaviour has two sides. The first can be an advantage because it saves time when making requests with different input parameters to discover possible combinations where the application may not respond as expected. Still, on the other hand, it doesn't validate unexpected behaviour of sequences, for example, the application will behave unexpectedly if it tries to use a resource created by another request before that request is executed, or if the order of operations is reversed.

With the result of this analysis and with the motivation to be able to extend the capacity of the RESTler, a set of properties was identified that can be validated through checkers, so the following

<sup>3</sup><https://www.doc.ic.ac.uk/~phjk/Publications/DynamicTopoSortAlg-JEA-07.pdf>

sections will detail how RESTler was extended to implement checkers to validate request sequencing, dependency between requests, hierarchy between operations or resources, idempotence and the correct use of HTTP Methods according to the endpoint.

### 5.3 OpenAPI Extensions for Related Operations

As stated in Section 2.2, the OAS can be used to specify RESTful APIs. This is a potent means of providing consumers with relevant information about the services. One of OAS's characteristics is that it lets us use the extensions<sup>4</sup> to construct or specify unique, non-standard fields for our API definition.

Additional data or metadata not covered by the default OpenAPI fields can be added using these extensions. To denote that the extensions are unique and not included in the standard specification, they are prefixed with **x-**, e.g., “*x-custom-extension*”, thus, taking into account this flexibility of OpenAPI, an extension to define the relationship between operations is proposed.

However, the Links typically look at each operation separately, from a consumer and producer perspective, ignoring any relationship between them or side effects that may occur when one of the operations is executed. Our proposal for introducing related operations to extend the standard OpenAPI is that we can help to alleviate this challenge once we can define and document their interaction modes among various operations.

This extension, known as *x-related-operations*, will be a new object in the OpenAPI. It's constituted by three fields: **operationId**, this field is of the type array of string [`string`], **relationship** field of type `string`, and **description**, this is a *optional* field of type `string`.

1. **operationId** – Using the operationId as a reference, we must indicate in this field which operations relate to the present operation; it is a standard OpenAPI attribute used to identify an operation and must be unique for every operation in the specification.
2. **relationship** – Describes the type of relationship (e.g., parent, child/nested, dependent). This field is important because the type of relationship defined will influence the test generation strategy. A relation defined as **parent** type means that this operation marked with this relationship doesn't depend on any other operation and can be run independently without needing pre-validation.

Once the relationships have been identified, as defined in the **operationId** field, it is possible to run tests to validate, for example, whether it is possible to change a resource before it has been created or whether by deleting a resource all the nested resources are also deleted. This type of scenario is covered by the checker “Dependencies-Hierarchy” described in Section 5.4.2.

A **child/nested** relation means that the operation in question can only be created if the operation on which it has a parent relationship has been executed previously. Therefore,

---

<sup>4</sup><https://spec.openapis.org/oas/v3.1.0#specification-extensions>

access to that resource can only occur if the parent resource exists. For example, to access a specific comment, e.g., `GET /api/blog/post/12/comments/99` made on a blog, it should only be possible if the blogpost entry exists, e.g., `GET /api/blog/post/12`.

When an operation is marked as **dependent**, it indicates that this operation will only exist when the parent operation exists and is similar to the **child/nested** relationship, but the difference is that in this, case, there will be no interaction for the operation to be triggered, i.e. once the parent operation is executed, the dependent operation is automatically triggered. For instance, an audit entry should be generated automatically and made accessible, e.g., `GET /api/blog/posts/{postId}/audit`, when a new blogpost entry is created, e.g., `POST /api/blog/post`, taking into consideration the `postId` that is returned in response to the blogpost creation.

3. **description** – An optional field used to describe or give more information to the consumers about the relation between operations. This field does not impact test generation, but it can be helpful as a mechanism to help understand the executed tests.

The Listing 18 shows how to use this extension in conjunction with the OpenAPI schema standard; some portions of the schema have been hidden to make them easier to understand. As we can see, a new object of type “**x-related-operations**” has been added to some operations. Starting with line 8, where we have the `POST /api/blog/post` with related operations, which is defined in the **operationId** field, which operations have a relationship with this operation, so the unique identifier of these operations has specified. In this case, we have four operations related to the current operation in this schema. The description field gives us an indication of the relationship established.

Other operations, such as `GET /api/blog/posts/{postId}/audit`, have a relationship of the type **dependent** with the operations `PUT DELETE /api/blog/post/{postId}`, or a **nested** relationship as specified in line 69. The fact that an operation has been referenced as having a relationship with another operation does not mean that we have to define a relationship with the primary operation in this operation.

```

1  ...
2  "paths": {
3    "/api/blog/posts": {
4      "post": {
5        "summary": "Create Post",
6        "operationId": "create_post_api_blog_posts_post",
7        "requestBody": {...},
8        "x-related-operations": {
9          "operationId": [
10           "update_blog_post_api_blog_posts_postId_put",
11           "get_blog_post_api_blog_posts_postId_get",
12           "delete_blog_post_api_blog_posts_postId_delete",
13           "get_blog_post_audits_api_blog_posts_postId_audit_get"
14         ],
15         "relationship": "parent",

```

```

16         "description": "When a blogpost is successfully created, the 'postId' value
           ↳ returned in the response can be used as a parameter in PUT
           ↳ /api/blog/posts/{postId}, GET /api/blog/posts/{postId} or DELETE
           ↳ /api/blog/posts/{postId} and the GET /api/blog/posts/{postId}/audit must
           ↳ return at least one object."
17     },
18     "responses": {...}
19 },
20 "get": {
21     "summary": "Get Blog Post",
22     "operationId": "get_blog_post_api_blog_posts__postId__get",
23     "parameters": [...],
24     "responses": {...}
25 }
26 },
27 "/api/blog/posts/{postId}": {
28     "put": {
29         "summary": "Update Blog Post",
30         "operationId": "update_blog_post_api_blog_posts__postId__put",
31         "parameters": [...],
32         "requestBody": {...},
33         "x-related-operations": {
34             "operationId": [
35                 "create_post_api_blog_posts_post"
36             ],
37             "relationship": "child",
38             "description": "When a blogpost is successfully updated, a new entry should be
39                 ↳ created on the Audit table."
40         },
41         "responses": {...}
42     }
43 },
44 "/api/blog/posts/{postId}/audit": {
45     "get": {
46         "summary": "Get Blog Post Audit",
47         "operationId": "get_blog_post_api_blog_posts__postId__audit__get",
48         "parameters": [...],
49         "x-related-operations": {
50             "operationId": [
51                 "create_post_api_blog_posts_post",
52                 "update_blog_post_api_blog_posts__postId__put",
53                 "delete_blog_post_api_blog_posts__postId__delete"
54             ],
55             "relationship": "dependent",
56             "description": "When a blogpost is successfully created, updated or deleted, a new
57                 ↳ entry should be created on the Audit table."
58         },
59         "responses": {...}
60     }
61 },
62 "/api/blog/posts/{postId}/comments/{commentId}": {
63     "get": {
64         "summary": "Get Blog Post Comments",
65         "operationId":
66             ↳ "get_blog_post_comments_api_blog_posts__postId__comments__commentId__get",
67         "parameters": [...],

```

```

65     "x-related-operations": {
66         "operationId": [
67             "create_post_api_blog_posts_post"
68         ],
69         "relationship": "nested",
70         "description": "It should be possible to access or create a comment if the
        ↪ blogpost, identified by the postId parameter, parent exists."
71     },
72     "responses": {...}
73 }
74 }
75 }
76 ...

```

Listing 18: OAS Extension x-related-operations

Although, at the moment, this extension allows us to describe relationships between operations in a more straightforward way than that used by Links, just by adding a few lines of code to the OAS, there is room for improvement, such as being able to specify multiple relationships, i.e. I might want to say that an operation can have a relationship of type “child” with an operation and a relationship of type “parent” with other operation. This would enable us to go further into the potential of test generation, therefore including a larger portion of the testable code.

## 5.4 Implementation of RESTler Checkers

As mentioned above, this study presents the implementation of a set of checkers to validate properties found in RESTful applications. These customised checkers will be integrated during the fuzzing process and managed by the RESTler Test Engine.

Some assumptions for the execution of the checkers:

- Each checker will run individually.
- Only valid sequences generated by the RESTler will be used by the checkers. Invalid sequences might lead to false positives by triggering errors that aren’t relevant.

### 5.4.1 Sequence Checker

As discussed earlier, the RESTler always tries to create a sequence of valid requests. A valid sequence refers to a series of API requests that correctly follow the logical order, e.g., a sequence where we create a resource, update it, and then delete it. For each request, it is ensured that the intended action was successfully performed, e.g., validate if the resource exists after creation (e.g., POST), reflects the updates after modification (e.g., PUT), and is no longer accessible after deletion (e.g., DELETE). Still, this checker aims to do the opposite, i.e., create an invalid sequence of requests, e.g., deleting or updating a resource before it has been created, and validate that the system reacts correctly, errors are reported as expected, and state-related bugs are identified.

To evaluate if the API management is correct in its state, we have to analyse the API request sequences and examine how the server responds. For instance, the API may create, edit, and delete resources and perform the sequence in reverse order. The main idea is to explore those sequences, which would uncover any irregularities in how API manages its state and thus be able to evaluate possible security issues.

Some scenarios, such as the sequences described below, can be validated:

- **Create** → **DELETE** → **Retrieve**: For this case, we will try to access a resource after it has been removed. Once more, the API should return a suitable error (e.g., 404 Not Found or 204 No Content).
- **Update** → **Retrieve**: This is an invalid sequence where an update or access to a resource is performed before it has been created. In this case, the API should return an appropriate status error code (e.g., 404 Not Found or 204 No Content).

Every method included in these sequences should be requested using GET requests, and the checker should validate the responses' authenticity against the system's intended state. It is noted here that the HTTP methods (GET, POST, PUT, and DELETE) should be used correctly in compliance with HTTP semantics. The server should use the correct status codes in response to the method applied and the state of the resource, whereby the correctness of each action must be checked against the respective method used.

To validate this approach, the following steps were done to evaluate if a custom checker could verify sequence state-related in a RESTful API:

1. **API State Transition**: Outline the valid and incorrect sequences in the state machine of the API. The default RESTler configuration will provide the valid sequence, and our checker will create the invalid sequence. To better illustrate this, Figure 5.4 shows an example of a valid sequence generated by RESTler.

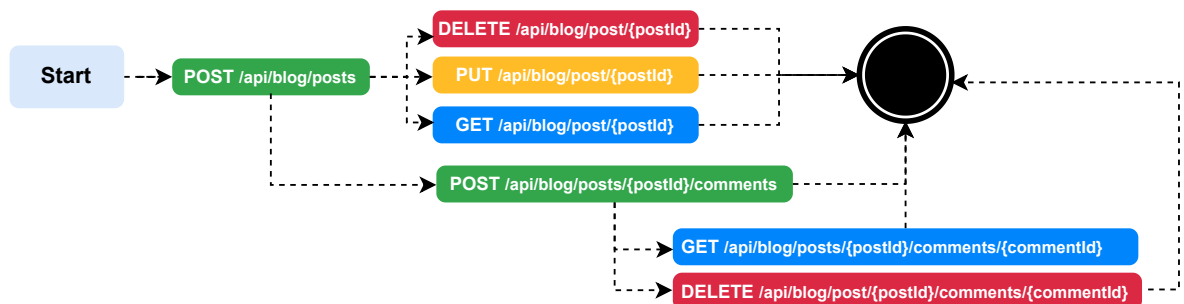


Figure 5.4: State Transition Valid Sequence

Figure 5.5 shows the invalid sequences created by our checker, where a resource is being updated (e.g., PUT) or deleted (e.g., DELETE) before it has been created, and the transition

of invalid states. In this case, the API should return a status code that matches the situation. Otherwise, this sequence should be reported as a bug.



Figure 5.5: State Machine Invalid Sequences

2. **Implement Invalid Sequence:** The following function was implemented as part of this checker, as seen in Listing 19 starting with some validations and restrictions. Some useful information is collected to help us analyse later, lines 12 and 17.

```

1  ...
2  def apply(self, rendered_sequence, lock):
3      if not rendered_sequence.valid or rendered_sequence.sequence.length <
4          ⇨ self.min_seq_length:
5          return
6
7      # Set the sequence and lock for the base implementation.
8      self._sequence = rendered_sequence.sequence
9      self._lock = lock
10
11     # Log and copy the original sequence methods.
12     original_methods = [req.method for req in self._sequence]
13     self._log(f"Original Sequence: {original_methods}")
14
15     # Reverse the sequence and methods
16     reversed_sequence = sequences.Sequence(list(reversed(self._sequence)))
17     reversed_methods = list(reversed(original_methods))
18     self._log(f"New Sequence: {reversed_methods}")
19
20     # Create a new sequence from the reversed sequence.
21     new_sequence = sequences.Sequence([])
22     for reversed_request in reversed_sequence:
23         new_sequence = new_sequence.__add__(reversed_request)
24     ...

```

Listing 19: Sequence Checker Initial Validation

As shown in Listing 20 in line 2, we have the valid (e.g., POST, GET), where we create the resource and retrieve later, and invalid sequences in line 3, (e.g., GET, POST), when we try to request a resource before it has been created.

3. **Generate Requests and Assertions:** As can be seen in Listing 21 in line 10, the checker will generate new requests to be sent to the server based on the new sequence produced in

```

1 ...
2 Original Sequence: ['POST', 'GET']
3 New Sequence: ['GET', 'POST']
4 New Request: /api/blog/posts/_READER_DELIM_api_blog_posts_post_id_READER_DELIM GET
5 Status Code: 404
6 New Request: /api/blog/posts POST
7 Status Code: 201
8 ...

```

Listing 20: Sequence Checker Output

Listing 19 in line 20. Following every run, an invalid sequence alters the original sequence, and the outcome of the request does not yield the desired result.

If an error is detected, it is added to the list of bugs (BugBuckets), *line 16*, to be analysed and reproduced in isolation.

```

1 ...
2 # Log and send each request in the new sequence.
3 for request in new_sequence:
4
5     request.headers_schema.append(HeaderParam("x-checker:",
6         ↪ "reverse-sequence"))
7     self._log(f"New Request: {request.endpoint} {request.method}")
8     seq = self._sequence + sequences.Sequence([request])
9
10    # Send the request and get the response
11    response, _ = self._render_and_send_data(seq, request)
12    self._log(f"Status Code: {response.status_code}",
13        ↪ print_to_network_log=False)
14
15    # Verify that the status code matches the expected status code
16    if response and self._rule_violation(seq, response, False):
17        self._print_suspect_sequence(seq, response)
18        BugBuckets.Instance().update_bug_buckets(seq,
19            ↪ response.status_code, origin=self.__class__.__name__)
20 ...

```

Listing 21: Sequence Checker Requests

4. **Execute and Log:** Once the checker is run and managed by the RESTler Test Engine, all interaction with the API, such as discrepancies or unexpected state transactions, is recorded in a log of its own, to be analysed later.
5. **Assessment:** In this stage, an analysis of the results is carried out, based on the logs generated previously, to identify state-related bugs and areas where the API does not conform to the expected behaviour.

The results obtained by running this checker in a fuzz-lean mode are shown in Table 5.1, where the original sequence is altered by an invalid sequence, the value shown in the result columns refers to the expected and obtained HTTP Status Code for each request individually.

Table 5.1: Analysing the Sequence Checker execution

| # | Original Sequence          | New Sequence               | Expected Results      | Achieved Results      |
|---|----------------------------|----------------------------|-----------------------|-----------------------|
| 1 | ['POST', 'GET']            | ['GET', 'POST']            | ['404', '201']        | ['200', '201']        |
| 2 | ['POST', 'POST']           | ['POST', 'POST']           | ['201', '409']        | ['201', '201']        |
| 3 | ['POST', 'PUT']            | ['PUT', 'POST']            | ['204', '201']        | ['204', '201']        |
| 4 | ['POST', 'DELETE']         | ['DELETE', 'POST']         | ['404', '201']        | ['404', '201']        |
| 5 | ['POST', 'POST', 'DELETE'] | ['DELETE', 'POST', 'POST'] | ['404', '201', '201'] | ['404', '201', '201'] |
| 6 | ['POST', 'POST', 'GET']    | ['GET', 'POST', 'POST']    | ['200', '201', '409'] | ['200', '201', '201'] |

As can be seen from the results shown in the table, out of a total of 6 test cases, half of the tests showed inconsistencies, tests 1, 2 and 6, showed an inconsistency between the expected result and the result obtained; in this case, it was expected that the API would return an HTTP Status Code 404 and 409, indicating record not found and duplicate data, but it returned a status code 20x, indicating that the operation was successful. When analysing the logs and the implementation, it became clear that the `POST /api/blog/post` operation does not perform any validation to check that the record being created already exists in the database.

#### 5.4.2 Dependencies-Hierarchy Checker

Resource hierarchies in APIs vary with different implementations because every application or API has its semantics for how they should behave. For example, the behaviour of deleting a resource with children or how dependencies should be specified and their preconditions ensured is not strictly defined by the HTTP protocol (RFC 9110 or its predecessors), leaving this up to each application.

The proposed extension can be used to address this type of situation since we can clearly specify the dependencies and hierarchy between operations using the fields available in this extension. So, this checker aims to validate that the hierarchy and dependencies inferred from analysing the OpenAPI schema (OAS) are being followed.

The Listing 23 shows how dependencies can be identified in the schema. Start by loading the grammar, line 5, where we have the schema enriched with the extension. On line 8, a function is called that will go through the entire schema and look at all the endpoints to see which ones have the declaration of the new extension.

Once the function is called, it iterates over all the paths (endpoints), line 14, which in turn iterate all the HTTP methods, line 16, for each identified endpoint. In line 17, another function is called to check recursively whether the extension exists in the schema. If the custom extension is found, then store the endpoint, method, and extension information as defined in the schema level, which will be used later on.

```

1  ...
2      self.grammar_path = Settings().get_checker_arg(self._friendly_name,
3      ↪ 'grammar_path')
4
5      # Load the Grammar schema
6      with open(self.grammar_path, 'r') as file:
7          grammar = json.load(file)
8
9      endpoints_with_extension = self.find_extension_in_endpoints(grammar,
10      ↪ self.extension_name)
11
12  ...
13
14  def find_extension_in_endpoints(self, schema, extension_name):
15      paths = schema.get('paths', {})
16      endpoints_with_extension = []
17
18      for endpoint, methods in paths.items():
19
20          for method, details in methods.items():
21              found = self.find_extension_in_schema(details, extension_name)
22              if found:
23                  endpoints_with_extension.append({
24                      'endpoint': endpoint,
25                      'method': method,
26                      'extension_value': found
27                  })
28
29      return endpoints_with_extension
30
31  def find_extension_in_schema(self, schema, extension_name):
32      if isinstance(schema, dict):
33          # Check if the extension exists at this level
34          if extension_name in schema:
35              return schema[extension_name]
36
37          # Otherwise, continue searching in nested dictionaries
38          for key, value in schema.items():
39              found = self.find_extension_in_schema(value, extension_name)
40              if found:
41                  return found
42
43      elif isinstance(schema, list):
44          # Search inside lists
45          for item in schema:
46              found = self.find_extension_in_schema(item, extension_name)
47              if found:
48                  return found
49
50      return None # Extension not found
51
52  ...

```

## Listing 22: Idempotence Checker Initial Setup

Figure 5.6 shows a sample of what is extracted by the process described in Listing 23.

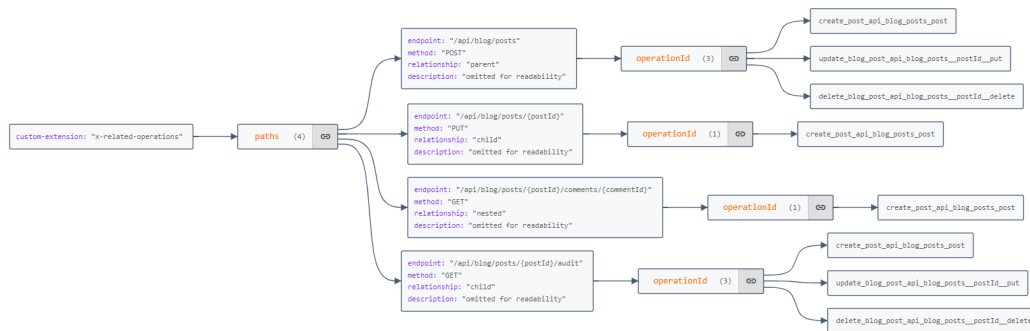


Figure 5.6: Related Operations Fragment

Once the dependencies have been identified, the next step is to check that the defined hierarchy in the OAS schema is correctly followed. As discussed in Section 5.3, the type of relationship specified would define the validation strategy to be followed. For example, If a relationship is defined as a parent and if the current request is a producer to create a new resource, then the operations identified as related cannot be executed before the current request, so the checker will try to execute these requests before executing the main request, to exploit potential flaws by not following the defined hierarchy. If any of these dependent operations return a 2xx status code, this indicates that the defined hierarchy has been violated and that a possible bug has been detected.

Listed below are the types of strategies carried out by this checker based on information extracted from the OAS schema:

- **Early access:** When a consumer from the current request is the operation being represented as related to the current operation, this strategy is used. In this case, a request is triggered to the related operation to try to access a resource that has not yet been created, and the expected result is that the application returns an error or any other status code other than 2xx Success.
- **Later access:** Contrary to the previously defined strategy, in this scenario, once the current request has been fully carried out, the related operations must be capable of being done successfully, resulting in a status code 2xx Success, depending on the HTTP method used by the related operation.
- **Side Effect:** This strategy is followed for operations that have a child relationship, where the related operation is only triggered when the parent operation is completed, in order to be able to validate that the result of an action has had an effect on the related operation.

Starting by taking the information that was collected earlier, during the identification of which operations are related, line 8 of Listing 23, the next step is to go to each of these operations and

check information which will be useful for defining the strategy, such as whether this operation is a consumer, the type of HTTP method used, the type of response, e.g., whether it returns an object or a list of objects.

This allows us to perform validations such as deleting all resources should a parent resource be deleted; hence, this kind of test can be verified as follows: try to access a child resource whose parent resource has been deleted; should this be done and the result is 2xx, it shows that this rule has not been followed and should be reported as a bug.

### 5.4.3 Idempotence Checker

In RESTful APIs <sup>5</sup>, idempotence of HTTP Methods refers to the ability for an action to be performed once or more times and yet yield the same outcome. HTTP requests like GET, PUT, DELETE, and HEAD are usually regarded as idempotent, while POST is not, as defined on RFC 2616<sup>6</sup> and shown in Table 5.2.

Table 5.2: HTTP Methods Idempotence and safeness classification

| HTTP Method | Idempotent | Safe |
|-------------|------------|------|
| GET         | yes        | yes  |
| HEAD        | yes        | yes  |
| PUT         | yes        | no   |
| DELETE      | yes        | no   |
| POST        | no         | no   |
| PATCH       | no         | no   |

An HTTP method is considered safe if its defined semantics are essentially read-only; i.e., the client does not request, and does not expect, any state change on the origin server as a result of applying a safe method to a target resource, as defined by [9].

Unless there has been an unsafe method, e.g., PUT or DELETE, requests performed in between, a GET request made frequently, such as `/api/blog/posts/{postId}`, should always return the exact resource representation without having any harmful effects.

For PUT method, repeated requests with the same parameter values should update the resource to the same state, e.g., PUT `/api/blog/posts/{postId}` and DELETE method in repeated sequences, e.g., DELETE `/api/blog/posts/{postId}` the first request should return a 200 Executed or 202 Accepted status code and the subsequent request should result in no changes and return a status code 204 No Content or 404 Not Found, any other status code should be reported as a bug.

As described in Subsection 2.4.3 about the RMM on Level 2, the HTTP methods are used according to their defined semantics; on this level, the idempotence becomes highly relevant once the consumers can rely on the current usage of the HTTP methods, ensuring they align with their HTTP semantics.

<sup>5</sup><https://restfulapi.net/idempotent-rest-apis/>

<sup>6</sup><https://www.rfc-editor.org/rfc/rfc2616#section-9.1>

An essential attribute of idempotence is its consistent, unchanging outcome without adverse repercussions. In a REST API where idempotence is not ensured<sup>7</sup>, we may have problems with duplication and inconsistency, for example, a *blogpost* table that, when adding a new record, is inserted into the audit table, the action carried out, e.g. create, update or delete the post entry; if it fails to be inserted into the audit table, we will have an inconsistency bug here.

Detecting this type of inconsistency is not a relatively simple task, given only the information in the standard OAS definition. However, using the relationship between operations, as proposed in Section 5.3, it is possible to identify that for a given operation, we will have an action that will affect another operation, in our case, the creation of a *blogpost* will trigger the creation of an audit entry.

The following steps were done to evaluate if a custom checker could verify idempotence in a RESTful API:

1. **Identify Idempotent Methods:** As shown in Table 5.2, not all methods are idempotent; in this checker, we will configure the checker to target GET, PUT, and DELETE methods, as defined in line 13 of Listing 23. The maximum number of requests, line 7, that will be made to the server is also specified and will be defined via a parameter in the checker's configuration file.

```

1  ...
2  class IdempotenceChecker(CheckerBase):
3      def __init__(self, req_collection, fuzzing_requests):
4          CheckerBase.__init__(self, req_collection, fuzzing_requests)
5
6          # The total number of requests to test
7          self._max_number_requests: int = None
8
9          # alias the log
10         self._log = self._checker_log.checker_print
11
12         # Define the idempotent methods
13         self._idempotent_methods = {'POST', 'PUT', 'DELETE'}
14
15     def apply(self, response, rendered_seq):
16         if not rendered_seq.valid:
17             return
18     ...

```

Listing 23: Idempotence Checker Initial Setup

<sup>7</sup>There are some mechanisms, such as using conditional-key, secondary-key, or idempotency-key, as a parameter in the message header, to allow the application to deal with idempotent situations and to avoid undesired effects. This study will not focus on whether this mechanism is being used.

To avoid unnecessary execution, we will start by eliminating execution in sequences that are not valid, as can be seen in line 16 of Listing 23. A sequence is considered invalid by RESTler if some requests don't have a 20x or 30x status code in the response.

2. **Create Requests:** Once the sequence is valid, we validate the method, line 3 on Listing 24, of the current request to ensure that we only validate methods that have been defined previously. We generate a new request using the same values for each identified method.
3. **Repeat Requests:** The same request will be sent as many times as defined, line 5, for each idempotent method in order to check that the application responds as expected.

```

1  ...
2  for request in self._sequence:
3      if request.method in self._idempotent_methods:
4          self._log(f"This method will be tested: {request.endpoint}"
                    ↪ {request.method} {request.metadata}
                    ↪ {request.endpoint_no_dynamic_objects}")
5          for i in range(1, self._max_number_requests + 1):
6              _send_request(request)
7      ...
8      continue
9  ...
10 def _send_request(request_to_send):
11     self._log("Sending successive request: \n"
12             ↪ f"{request_to_send.definition}",
13             ↪ print_to_network_log=False)
14     seq = self._sequence + Sequence(request_to_send)
15     response, _ = self._render_and_send_data(seq, request_to_send)
16     ...

```

Listing 24: Idempotence Requests

4. **Compare Responses:** Compare responses from repeated requests:
  - GET: The resource representation should remain the same. If a resource changes after successive requests, it should be checked whether there have been any other requests that may have influenced this resource, this can be done by looking at the sequence of requests that have been made to the server, if no other interaction is detected, it should be reported as a bug.
  - PUT: The resource should be updated only once, and subsequent PUT should yield the same state. For requests of this type, the initial state must be compared with the following states, any change detected, either in the status code returned or in the message body, should be reported as a possible bug.
  - DELETE: After the resource is deleted, subsequent DELETE requests should indicate the resource no longer exists, e.g., 204, or return a consistent status, e.g., 404, any other status code should be reported as a potential bug.

```

1  ...
2      # Compare current response with the stored response
3      if response.status_code != self.previous_response.status_code:
4          error_msg = (
5              f"Idempotency check failed: {response.request.method}
6              ↳ {response.request.endpoint}. "
7              f"Expected status code {self.previous_response.status_code}, "
8              f"got {response.status_code}"
9          )
10
11     if response.body != self.previous_response.body:
12         error_msg = (
13             f"Idempotency check failed: {response.request.method}
14             ↳ {response.request.endpoint}. "
15             f"Response body differs."
16         )
17     ...

```

Listing 25: Idempotence Validate Response

5. **Report Non-Idempotence:** If any discrepancies are found, e.g., status code incompatible, body message with different values, this request should be reported as a violation of idempotence.

```

1  ...
2      if response and response.has_bug_code():
3          self._print_suspect_sequence(new_seq, response)
4          BugBuckets.Instance().update_bug_buckets(new_seq,
5          ↳ response.status_code, origin=self.__class__.__name__)
6  ...

```

Listing 26: Idempotence Report Bug

#### 5.4.4 Method Not Allowed Checker - MnA

Web API may be vulnerable to security flaws if HTTP methods are mishandled. For instance, unauthorised data deletion may occur if an endpoint unintentionally permits a DELETE request when it shouldn't. Many industry standards and guidelines for API development (such as OWASP<sup>8</sup> API Security Top 10) emphasise correctly handling HTTP methods. Method Not Allowed tests improve web API security by detecting and reducing these threats.

Testing for HTTP status code 405 ('**Method Not Allowed**') response ensures that a RESTful API adheres to best practices and operates securely and predictably. This section explains the implementation of this checker and its motives in depth.

<sup>8</sup>[https://cheatsheetseries.owasp.org/cheatsheets/REST\\_Security\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html)

As described in Subsection 2.4.3, at RMM Level 2, a REST API is expected to fully use HTTP methods (verbs) such as GET, POST, PUT, PATCH, DELETE, etc, and each method should be used according to its intended purpose.

This checker allows verifying whether the restrictions imposed by an API on inappropriate HTTP methods for any resource are respected. Which areas where APIs should be changed to allow or disallow some particular method(s) so that they can reach and maintain RMM Level 2 would also be indicated. For instance:

- A GET request on a resource might be allowed, but a POST request might not be (**'405 Method Not Allowed'**).
- A PUT request might not be allowed on a resource intended to be immutable.

As we have already emphasised, the RMM Level 2 requires that HTTP methods be used appropriately following REST principles, where each endpoint should only accept HTTP methods that make sense for the resource it represents. The MnA checker directly validates that service at RMM Level 2 is enforcing these rules. Thus, if an API correctly returns a **'405 Method Not Allowed'** status code for inappropriate HTTP methods, it gives us an indication that it complies with RMM Level 2 and respects the semantics of HTTP methods.

For example, the API should return a (**'405 Method Not Allowed'**) if a request was made for a resource using a request method not supported by that resource, e.g., using GET on a form that requires data to be presented via POST or using PUT on a read-only resource. As described in Section 5.1, we have the following endpoint at `/api/blog/posts/{postId}`, on the RMM Level 2, this API should allow:

- GET `/api/blog/posts/{postId}` to retrieve a Blog Post.
- PUT `/api/blog/posts/{postId}` to update a Blog Post.
- DELETE `/api/blog/posts/{postId}` to delete a Blog Post.

If the application receives a request POST `/api/blog/posts/{postId}`, it must return **'405'** HTTP status code, because POST is not meant for updating or deleting from this endpoint. This test will validate whether the API fulfils HTTP method semantics on the RMM level 2 maturity scale.

A matrix representing the relationships between the HTTP Method used and the respective properties, **Consumer** and **Body Schema**, is displayed in Table 5.3. This relationship is essential to creating specific test cases during the “swapping” process.

In this context, the **Consumer** property refers to a dynamic object in the endpoint, e.g., GET `/api/blog/posts/{postId}` the `{postId}` is a consumer that was produced for another endpoint, e.g., POST `/api/blog/posts`, where the message body has an attribute identified as `{postId}`.

Table 5.3: HTTP Method Composition

| HTTP Method | Consumer  | Body Schema |
|-------------|-----------|-------------|
| PUT         | Mandatory | Mandatory   |
| POST        | N/A       | Mandatory   |
| DELETE      | Mandatory | N/A         |
| GET         | Optional  | N/A         |

The Body Schema refers to the **message body**<sup>9</sup> in HTTP terms, and according to RFC7231, section 4.3.8<sup>10</sup>:

“A client must not send a message body in a TRACE request.”

This technically means that all other methods, e.g., GET, HEAD, POST, PUT, DELETE, can include the **message body** during the request, but it’s more common to find this property in POST, PUT and PATCH, which is why Table 5.3 shows this property as mandatory.

The Table 5.4 shows a sequence of combinations that were made during the fuzz-lean execution mode, where we exchanged the original HTTP method for another random method; the expected result and the result obtained are also shown. RESTler logs served as the basis for this investigation.

Table 5.4: Analysing the MnA checker execution

| # | Original HTTP Method            | Swapped HTTP Method             | Expected Result | Achieved Result |
|---|---------------------------------|---------------------------------|-----------------|-----------------|
| 1 | POST /api/blog/posts            | PUT /api/blog/posts             | Fail            | Fail            |
| 2 | GET /api/blog/posts/{postId}    | POST /api/blog/posts/{postId}   | Fail            | Fail            |
| 3 | PUT /api/blog/posts/{postId}    | DELETE /api/blog/posts/{postId} | Fail            | Pass            |
| 4 | DELETE /api/blog/posts/{postId} | POST /api/blog/posts/{postId}   | Fail            | Fail            |
| 5 | GET /api/blog/posts/{postId}    | PUT /api/blog/posts/{postId}    | Fail            | Fail            |
| 6 | DELETE /api/blog/posts/{postId} | PUT /api/blog/posts/{postId}    | Fail            | Fail            |
| 7 | PUT /api/blog/posts/{postId}    | POST /api/blog/posts/{postId}   | Fail            | Fail            |

When analysing the results obtained, in which there was an inconsistency between the expected result and the observed one, test case id 3, it can be seen that the PUT and DELETE operations share the same path, e.g., /api/blog/posts/{postId}, even though at specification level they have other properties that differentiate them, for example PUT has a body schema, while DELETE does not have this property. From this point of view, the request made should fail because the original request was to update a record; however, the request sent was a DELETE, which in a real scenario can have unwanted effects.

Testing to restrict methods is essential for validating the proper implementation of HTTP methods by an API, which is one of the central requirements under RMM level 2. In this way, it guarantees that its adherence to REST principles is observed by validating the adequate usage of these verbs. Thus, this ensures that at RMM Level 2, an API can be considered a genuinely RESTful service.

<sup>9</sup><https://www.rfc-editor.org/rfc/rfc7230#section-3.3>

<sup>10</sup><https://www.rfc-editor.org/rfc/rfc7231#section-4.3.8>

## Chapter 6

# Evaluation

This chapter presents the validation process for the custom checkers developed to enhance API testing using RESTler. We evaluate the effectiveness of these checkers in identifying issues such as sequence violations, idempotency violations, dependencies hierarchy handling and method misuse. The validation is based on experiments and comparisons against the normal RESTler without additional configuration and the RESTler with extended checkers.

### 6.1 Evaluation Process

The validation process was conducted on a RESTful API developed for a sample Pet Store<sup>1</sup> system. The API supports creating, updating, and deleting pets, stores and users. It has dependencies between resources (e.g., an order cannot be placed if no pet is created). The tests were performed using RESTler with custom checkers developed as part of this thesis, and the original OpenAPI<sup>2</sup> schema was enriched with the proposed extension. Schemathesis, another tool that was identified and described in Section 3.2, will be used for comparative purposes.

#### 6.1.1 Test Environment Setup

This subsection details the test environment setup used for validating the custom checkers. The environment includes the Pet Store API, deployed using Docker and RESTler in simple and extended modes with custom checkers and Schemathesis with all built-in Checks enabled.

##### 6.1.1.1 Environment Configuration

The entire test environment was hosted on a personal computer with the following specifications:

- **Operating System:** Windows 11 Pro
- **Processor:** i5-8350U CPU @ 1.70GHz, 4 Cores, 8 Logical Processors

---

<sup>1</sup><https://github.com/swagger-api/swagger-petstore>

<sup>2</sup><https://petstore3.swagger.io/api/v3/openapi.json>

- **RAM:** 16 GB
- **Docker version:** 27.2.0
- **Python version:** 3.9.10
- **RESTler version:** latest release available in GitHub<sup>3</sup>

This configuration ensured that the Pet Store API and RESTler could be run efficiently, providing consistent results for the validation process.

### 6.1.1.2 Pet Store API Setup Using Docker

The Pet Store API, a sample OpenAPI specification, was deployed locally using Docker. Docker was chosen for its ease of use, portability, and ability to set up isolated environments quickly. The following steps were taken to set up the Pet Store API:

1. Ensured that Docker was installed and running locally.

```
C:\restler-fuzzer\validation\petstore\working-dir>docker -v
Docker version 27.2.0, build 3ab4256
```

2. The official Pet Store API image was pulled from DockerHub using the following command:

```
C:\restler-fuzzer\validation\petstore\working-dir>docker build -t
↪ swaggerapi/petstore3:unstable .
```

3. The Pet Store API was then started with the following Docker command:

```
C:\restler-fuzzer\validation\petstore\working-dir>docker run -d -p 8080:8080
↪ swaggerapi/petstore3
```

This command runs the Pet Store API on port 8080 of the local machine.

4. Once running, the API was accessible at <http://localhost:8080/>.

---

<sup>3</sup><https://github.com/microsoft/restler-fuzzer>

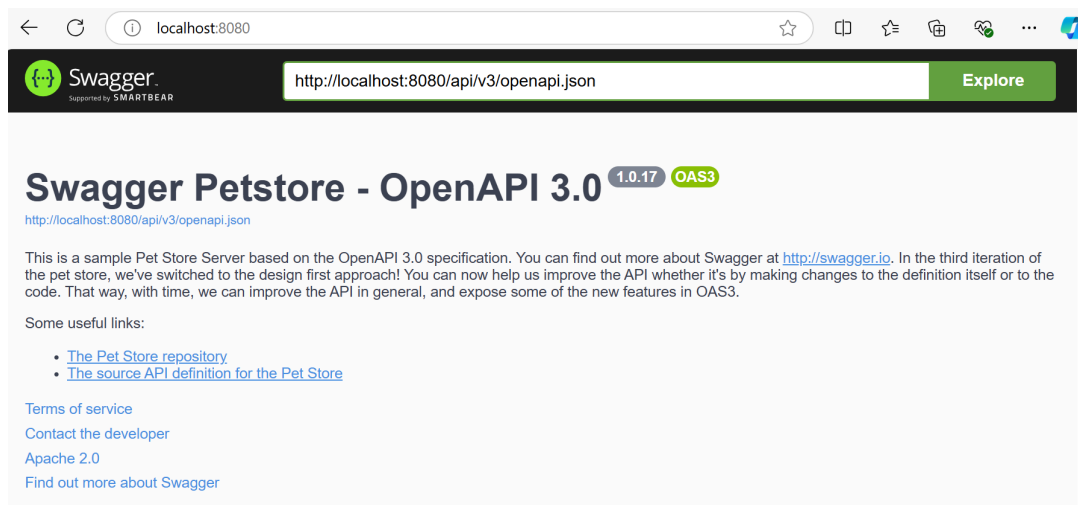


Figure 6.1: Pet Store OpenAPI

### 6.1.1.3 RESTler in Simple Mode

In a simple mode, e.g., RESTler has been installed, and no settings have been changed. RESTler was run without any custom checkers, utilizing its default capabilities to test the API for common issues such as invalid parameters, missing parameters or error 500.

The setup for RESTler in simple mode was as follows:

1. RESTler was installed by downloading its latest release from the official repository.
2. RESTler was configured to use the Pet Store API's OpenAPI specification to generate the request grammar:

```
C:\restler-fuzzer\validation\petstore\working-dir>restler compile --api_spec
↳ openapi.json
Starting task Compile...
Task Compile succeeded.
Collecting logs...
```

3. RESTler was then executed in Test mode: As described in Section 4.2.2, when running in Test mode, RESTler tried to execute all the operations at least once successfully.

```
C:\restler-fuzzer\validation\petstore\working-dir>restler.exe test --grammar_file
↳ .\Compile\grammar.py --dictionary_file .\Compile\dict.json --settings
↳ .\Compile\engine_settings.json --no_ssl
Starting task Test...
Using python: 'python.exe' (Python 3.9.10)
Request coverage (successful / total): 13 / 19
Attempted requests: 15 / 19
Bugs were found!
Bug buckets:
main_driver_500: 1
See 'coverage_failures_to_investigate.txt' to investigate API coverage.
Task Test succeeded.
Collecting logs...
```

4. RESTler was then executed in fuzzing mode. This command runs the default fuzzing tests against the Pet Store API, sending various combinations of requests to check for issues such as server error (500), parameter validation and resource creation/deletion.

```
C:\restler-fuzzer\validation\petstore\working-dir>restler.exe fuzz-lean
↪ --grammar_file .\Compile\grammar.py --dictionary_file .\Compile\dict.json
↪ --settings .\Compile\engine_settings.json --no_ssl
Starting task FuzzLean...
Using python: 'python.exe' (Python 3.9.10)
Request coverage (successful / total): 13 / 19
Attempted requests: 15 / 19
Bugs were found!
Bug buckets:
InvalidDynamicObjectChecker_20x: 2
InvalidValueChecker_500: 4
PayloadBodyChecker_500: 53
main_driver_500: 3
See 'coverage_failures_to_investigate.txt' to investigate API coverage.
Task FuzzLean succeeded.
Collecting logs...
```

#### 6.1.1.4 RESTler with Extended Checkers

To validate the custom checkers developed in this thesis, RESTler was run in an extended mode, utilizing additional checkers for specific issues such as idempotency, dependency violations, and method misuse. The custom checkers were integrated into RESTler in the following way:

1. The RESTler Test Engine was extended to include the custom checkers, which were implemented by modifying RESTler's request sequence generation and response validation components.
2. Similar to standard mode, the grammar was generated using the Pet Store API specification but with additional parameters to add custom checkers and the extension proposed:

```
C:\restler-fuzzer\validation\petstore\working-dir>restler compile --api_spec
↪ openapi_withextension.json
Starting task Compile...
Task Compile succeeded.
Collecting logs...
```

3. **Run RESTler with Custom Checkers:** The extended version of RESTler was then executed in fuzzing mode with the custom checkers enabled: This command allows RESTler to perform extended checks, including idempotency validation, invalid request sequences, and other custom operations developed as part of this thesis.

```
C:\restler-fuzzer\validation\petstore\working-dir>restler fuzz-lean
↪ --grammar_file .\Compile\grammar.py --dictionary_file .\Compile\dict.json
↪ --settings .\Compile\engine_settings.json --no_ssl
Starting task FuzzLean...
Using python: 'python.exe' (Python 3.9.10)
Request coverage (successful / total): 15 / 19
Attempted requests: 15 / 19
```

```

Bugs were found!
Bug buckets:
InvalidDynamicObjectChecker_20x: 2
InvalidValueChecker_500: 5
PayloadBodyChecker_500: 55
main_driver_500: 5
SequenceChecker_20x: 3
IdempotenceChecker_20x: 5
MethodNotAllowedChecker_40x: 11
See 'coverage_failures_to_investigate.txt' to investigate API coverage.
Task FuzzLean succeeded.
Collecting logs...

```

### 6.1.1.5 Schemathesis with built-in Checks

The Schemathesis was installed locally via pip: `python -m pip install schemathesis`

```

PS C:\Users\gedsi> schemathesis --version
schemathesis, version 3.27.0

```

After installation, it was run using the Pet Store API schema without any changes. The **F** at the end of each endpoint means that Schemathesis was unable to execute the operation in question successfully, i.e., it got an HTTP Status code 2x in the response:

```

PS C:\Users\gedsi> schemathesis run --checks all
↪ http://localhost:8080/api/v3/openapi.json --report
===== Schemathesis test session starts =====
Schema location: http://localhost:8080/api/v3/openapi.json
Base URL: http://localhost:8080/api/v3
Specification version: Open API 3.0.2
Random seed: 174124902661914435169498007824803549071
Workers: 1
Collected API operations: 19
Collected API links: 0
Report to Schemathesis.io: ENABLED
API probing: SUCCESS
Schema analysis: SUCCESS

POST /api/v3/pet F [ 5%]
PUT /api/v3/pet F [ 10%]
GET /api/v3/pet/findByStatus F [ 15%]
GET /api/v3/pet/findByTags . [ 21%]
DELETE /api/v3/pet/{petId} F [ 26%]
GET /api/v3/pet/{petId} . [ 31%]
POST /api/v3/pet/{petId} F [ 36%]
POST /api/v3/pet/{petId}/uploadImage F [ 42%]
GET /api/v3/store/inventory F [ 47%]
POST /api/v3/store/order F [ 52%]
DELETE /api/v3/store/order/{orderId} F [ 57%]
GET /api/v3/store/order/{orderId} . [ 63%]
POST /api/v3/user F [ 68%]
POST /api/v3/user/createWithList F [ 73%]
GET /api/v3/user/login F [ 78%]
GET /api/v3/user/logout . [ 84%]
DELETE /api/v3/user/{username} F [ 89%]

```

```

GET /api/v3/user/{username} F [ 94%]
PUT /api/v3/user/{username} F [100%]

===== SCHEMA ANALYSIS =====
Schema Analysis completed successfully.
Analysis took: 0.12ms
The following extensions have been applied:
- 3 schema patches
===== SUMMARY =====
Performed checks:
not_a_server_error          365 / 400 passed      FAILED
status_code_conformance    333 / 400 passed      FAILED
content_type_conformance    400 / 400 passed      PASSED
response_headers_conformance 400 / 400 passed      PASSED
response_schema_conformance 393 / 400 passed      FAILED
===== 4 passed, 15 failed in 90.72s =====

```

### 6.1.2 Metrics Collected During Validation

The following metrics were collected from standard RESTler logs, Grafana, and the extension that was created to RESTler as stated in Section 5.2 to analyse the effectiveness of the custom checkers that were implemented when the Pet Store API was being used. Grafana was used to collect and visualize metrics during the validation process and was set up to run locally using Docker. The Schemathesis metrics were collected via the standard report generated at the end of the test run.

- **Number of Issues Detected:** The comparison of issues found by standard RESTler checkers versus custom checkers can be seen in Table 6.1, which shows the total number of issues detected by each custom checker, e.g., idempotency violations, invalid sequences, method misuse and dependencies violations.

In a RESTler Simple Mode, the following checkers are available by default: InvalidDynamicObjectChecker\_20x, InvalidValueChecker\_500 and PayloadBodyChecker\_500. When run in Extended Mode, the following customized checkers are added: SequenceChecker\_20x, DependenciesHierarchyChecker\_20x, IdempotenceChecker\_20x and MethodNotAllowedChecker\_40x.

Table 6.1: Issues Reported

| Checker                          | RESTler Simple Mode | RESTler Extended Mode |
|----------------------------------|---------------------|-----------------------|
| InvalidDynamicObjectChecker_20x  | 2                   | 2                     |
| InvalidValueChecker_500          | 4                   | 4                     |
| PayloadBodyChecker_500           | 53                  | 53                    |
| SequenceChecker_20x              | -                   | 3                     |
| DependenciesHierarchyChecker_20x | -                   | 0                     |
| IdempotenceChecker_20x           | -                   | 5                     |
| MethodNotAllowedChecker_40x      | -                   | 11                    |
| <b>TOTAL</b>                     | <b>59</b>           | <b>78</b>             |

While RESTler could report a significant number of errors, Schemathesis reported only 6 considered server errors and 13 schema conformance errors, as seen in Figure 6.2.

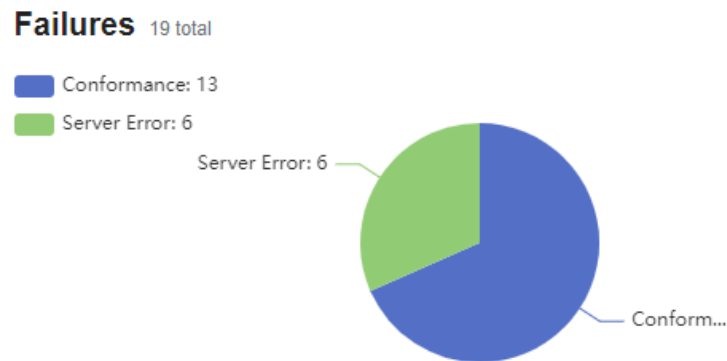


Figure 6.2: Failures Reported by Schemathesis

These errors are distributed as follows:

- 9 - related to **Undocumented HTTP status code**, e.g., received: 500 and documented: 200, 400, 404.
  - 6 - related to **Server error**, e.g., an unhandled exception at application level.
  - 3 - related to **Response violates schema**, e.g., a required property was not sent during the request.
  - 1 - related to **JSON deserialization error**, e.g., an expected value not found in the request.
- **Execution Time:** Since the collection of endpoints had only 19 operations, the execution time was quick, about 1.5 hours before all the operations were completed and exercised at least 20 times, as defined in the RESTler configuration file to cover as many test scenarios as possible. Resource consumption was greatly reduced during this time, as seen in Figure 6.3, which indicates that even with more validations to do, RESTler shows good resource utilisation, which its modular architecture can explain.
    - Total execution time for fuzzing and validation: 90 minutes.
    - Idempotence Checker runtime: 30 minutes.
    - Sequence Checker runtime: 25 minutes.
    - Dependencies-Hierarchy Checker runtime: 15 minutes
    - Method Not Allowed Checker runtime: 20 minutes.

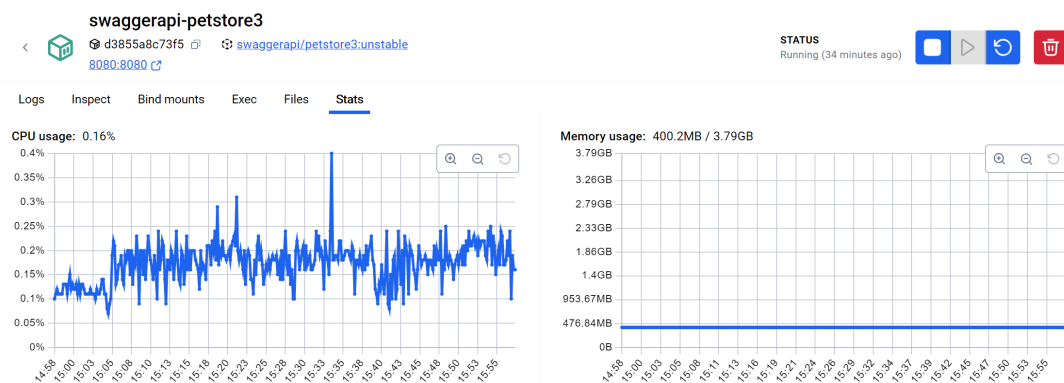


Figure 6.3: Resource Consumption

The RESTler identified issues of type 500 within the first few seconds of execution. The first errors indicating possible application bugs, which returned a status code of type 20x, such as a resource was deleted and then could be retrieved (GET), began after 10 minutes of execution.

- **API Endpoints Covered:** The percentage of API endpoints tested by the checkers, indicating how thoroughly the API was tested.
  - Total number of endpoints in the Pet Store API: 19.
  - Request coverage (successful / total): **13 / 19** when running using the RESTler Simple Mode, 68% coverage.
  - Request coverage (successful / total): **11 / 19** when running using the RESTler Extended, 57% coverage.

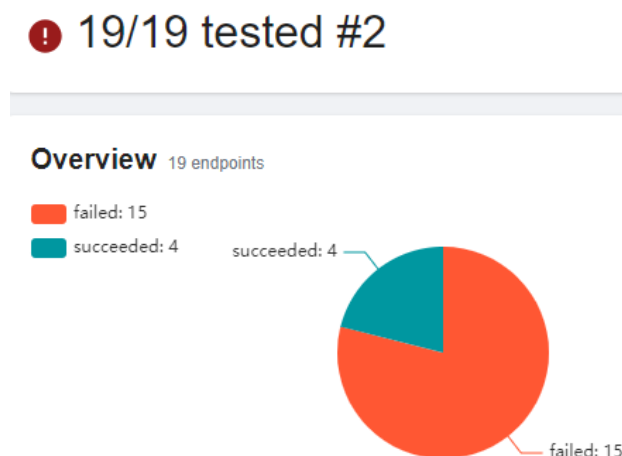


Figure 6.4: Coverage Reported by Schemathesis

The coverage achieved by Schemathesis was below that presented by RESTler, managing to execute only 4 endpoints out of 19 successfully.

- **Sequences Tested (Valid and Invalid):** This metric shows the number of valid and invalid sequences generated during the entire time the RESTler was running.
  - The number of valid sequences tested: 19 valid sequences, covering typical operations such as `POST`, `GET`, `PUT`, and `DELETE`.
  - The number of invalid sequences tested: 7 sequences, such as `DELETE` before `POST` or referencing non-existent resources.
- **Idempotence Violations Detected:** The number of idempotence violations found when repeating operations like `POST` or `PUT`, as defined in Subsection 5.4.3.
  - 5 idempotence violations detected, including repeated `PUT` and `DELETE` requests not returning the same result or status code.
- **Dependencies-Hierarchy Violations Detected:** This metric helps us to assess whether the API enforces the correct sequence of requests, ensuring that operations respect the defined hierarchy or dependencies between them, as specified in OpenAPI schema using the proposed extension.
  - 0 violations detected.
- **Method Not Allowed Violations Detected:**
  - 11 the number of method misuse violations found when an HTTP Verb is exchanged for another, such as a `POST` swapped to a `GET`, or `DELETE` swapped for a `PUT`.
- **HTTP Status Codes Observed:** The distribution of HTTP status codes observed during the testing, e.g., 200 OK, 400 Bad Request, 404 Not Found, 405 Method Not Allowed, 414 URI Too Long and 500 Internal Server Error.

Table 6.2: Status Code Observed

| HTTP Status Code | RESTler Simple Mode | RESTler Extended Mode |
|------------------|---------------------|-----------------------|
| 200              | 630                 | 1246                  |
| 400              | 120                 | 2800                  |
| 404              | 22                  | 7                     |
| 405              | 11                  | 7                     |
| 414              | 0                   | 11                    |
| 500              | 850                 | 2011                  |
| <b>TOTAL</b>     | <b>2713</b>         | <b>6082</b>           |

- **False Positives:** This metric helps us assess the custom checker precision, where the number of false positives generated by each checker is defined as cases where the checker incorrectly flags an issue.
  - **Idempotence Checker:** During the execution, 1 false positive was pointed out, where a repeated `PUT` request was incorrectly flagged as an error.

- **False Negatives:** The number of false negatives represents issues not detected by the checker but should have been; this is useful to evaluate the checker's coverage.
  - **Idempotence Checker:** 2 false negatives, where issues with repeating `DELETE` requests were missed.
  - **Sequence Dependency Checker:** 1 false negative, where a `DELETE` operation before `POST` was not flagged.

## 6.2 Discussion

It is time to discuss the metrics that have been collected and verify that **RQ1** and **RQ2** defined in Section 1.2 are answered with the proposed extension to OpenAPI and the implementation of custom checkers in RESTler. Starting by analysing the coverage of the API by each of the executions, it is notable that even with the standard use of RESTler, it is not possible to have 100% coverage of the endpoints, yet the percentage of coverage achieved was sufficient to detect errors in the API, as will be discussed later.

As can be seen in Table 6.1, which shows the results of the bug analysis, when the RESTler is run without any further setup, a large number of problems with status code 500 and a few with status code 200 can be recognized by default. When RESTler is run in Extended Mode, the number of bugs detected by the standard checkers has not changed, indicating that the proposed extension has not affected these checkers. Thus, it is necessary to carry out an extra analysis to validate how the information injected into the schema could be used to improve the validations carried out by these checkers. Still, as the main objective was not to enable the standard checkers to take advantage of this extension, this in-depth analysis was not conducted.

However, on the other hand, the proposed checkers detected 19 bugs in total distributed among the different checkers. Of this total, one was considered a false positive, which brings us to 18 real bugs. Based on the logs generated by RESTler, it was possible to identify in which cases Sequence Checker pointed out 3 bugs. During the execution phase of the RESTler in fuzz-lean and fuzz mode, a significant number of valid and invalid sequences were generated, such as:

- **Valid Sequence 1: Add a new pet, create an order, then retrieve the order by ID**

- `POST /api/v3/pet` (Add a new pet)
- `POST /api/v3/store/order` (Place an order for the pet)
- `GET /api/v3/store/order/{orderId}` (Retrieve order by ID)

- **Valid Sequence 2: Create a user, place an order, and then retrieve the order**

- `POST /api/v3/user` (Create a user)
- `POST /api/v3/store/order` (Place an order for a pet)
- `GET /api/v3/store/order/{orderId}` (Retrieve the order)

- **Valid Sequence 3: Create a user, login, retrieve the user data, and logout**

- POST /api/v3/user (Create a user)
- GET /api/v3/user/login (Log the user in)
- GET /api/v3/user/{username} (Retrieve the user's data)
- GET /api/v3/user/logout (Log the user out)

- **Invalid Sequence 1: Retrieve an Order Without Placing It**

- GET /store/order/{orderId} (Attempt to retrieve an order that has not been placed)
- DELETE /store/order/{orderId} (Attempt to delete a non-existent order)

- **Invalid Sequence 2: Log Out a User Without Logging In**

- GET /user/logout (Attempt to log out a user without logging in first)
- POST /user (Create a user after attempting to log out)
- GET /user/login (Logs user into the system)

Each of these invalid sequences fails because they attempt to perform operations on resources that are either missing or improperly initialized due to unmet dependencies between API operations as defined in OpenAPI schema. However, the checker implemented was able to create invalid sequences in which the expected result was to receive a response with a 404 or 400 status code, and the observed result was a 20x status code, indicating that the operations had been performed successfully. Figure 6.5 shows an example of a bug reported by the Sequence checker.

This sequence starts with a GET to a resource that doesn't exist, receiving a 404 as expected, and then attempting to place an order for a pet using a non-existent `petId`, and finally trying to retrieve the placed order, and for these two operations, it received a status code of 200 OK, while the expectation was to fail, since the pet doesn't exist.

```
-> GET /api/v3/pet/9999 HTTP/1.1\r\nAccept: application/json\r\nHost: localhost:8080\r\n\r\n",
! producer_timing_delay 0
! max_async_wait_time 0
PREVIOUS RESPONSE: HTTP/1.1 404 Not Found\r\nDate: Fri, 20 Sep 2024 16:41:32 GMT\r\nAccess-Control-Allow-Origin: *\r\nAccess-Control-Allow-Methods: GET, POST, DELETE, PUT\r\nAccess-
-> POST /api/v3/store/order HTTP/1.1\r\nAccept: application/json\r\nHost: localhost:8080\r\nContent-Type: application/json\r\n\r\n{"id":10,\r\n"petId":9999,\r\n"quantity":1}
! producer_timing_delay 0
! max_async_wait_time 0
PREVIOUS RESPONSE: HTTP/1.1 200 OK\r\nDate: Fri, 20 Sep 2024 16:41:33 GMT\r\nAccess-Control-Allow-Origin: *\r\nAccess-Control-Allow-Methods: GET, POST, DELETE, PUT\r\nAccess-Cont
-> GET /api/v3/store/order/10?injected_query_string=123 HTTP/1.1\r\nAccept: application/json\r\nHost: localhost:8080\r\n\r\n
! producer_timing_delay 0
! max_async_wait_time 0
PREVIOUS RESPONSE: HTTP/1.1 200 OK\r\nDate: Fri, 20 Sep 2024 16:41:34 GMT\r\nAccess-Control-Allow-Origin: *\r\nAccess-Control-Allow-Methods: GET, POST, DELETE, PUT\r\nAccess-Cont
```

Figure 6.5: Sequence Checker - Bug Reported

Other checkers, such as IdempotenceChecker and MethodNotAllowedChecker, could also detect errors in the Pet Store API. MethodNotAllowedChecker had the highest number of bugs detected, representing 11 out of a total of 18. The Dependencies-Hierarchy checker was unable to detect any bugs. The only way to reproduce this type of error would be to change the original implementation code. Unfortunately, we still cannot do this due to the constraints of setting up the local API development environment.

Most of the errors reported can be corrected at the implementation level, such as before creating an order `POST /api/v3/store/order`, validating that the pet included in the order already exists, or not allowing the creation of a pet with a status other than those defined, for example at schema level it is defined that status values that need to be considered are: `available`, `pending` and `sold`, but it was possible to update a pet, e.g., `PUT /api/v3/pet`, using a random status, like `reserved`.

The results show that the custom checkers effectively detect common API issues, particularly in method misuse, sequence dependency and idempotency scenarios, which leads us to answer the **RQ1**. Compared to existing RESTler checkers, the custom implementations demonstrated improved detection rates for specific operation-related errors, such as method misuse. However, some false positives were encountered, particularly in cases involving complex multi-resource relationships, indicating that further refinement is needed to reduce unnecessary alerts.

The extension proposed for OpenAPI also proved to be efficient when used with RESTler to support the identification of dependencies between operations, which explains the significant increase in HTTP Status Codes observed, which means more requests executed, as can be seen in Table 6.2. When revisiting the **RQ2**, the answer to **RQ2** is that using the proposed extension in combination with the proposed checkers proved effective. In contrast, the standard checkers were not as effective. Still, it is worth stating that with the information provided by this extension, it is possible to create checkers to validate more specific properties of the API.

## 6.3 Limitations

One limitation of the validation process was the relatively small number of real-world APIs tested. Additionally, the checkers could not handle highly complex API relationships as effectively as expected, leading to some false positives. Future work could improve handling complex API scenarios and extend the validation to a wider range of real-world APIs.

## Chapter 7

# Conclusions and Future Work

In this dissertation's final chapter, we will present our conclusions based on the conducted research. Later on, we will discuss potential future research directions for this study.

### 7.1 Conclusions

In recent years, there have been many studies in this area, but there is still a lot of potential for further research. With the increasing use of cloud services accessed via RestAPI and the tendency for applications to be built on microservices architecture, this type of study is becoming even more important. These studies have revealed significant insights and advancements in enhancing the robustness and reliability of web applications.

With this in mind, we believe that our study can contribute to the exploration of creating automated tests to validate such properties that were presented in Chapter 5, using RESTler checkers and the OAS extension proposed.

### 7.2 Future Works

As mentioned earlier, there is much room for improvement when testing REST APIs. Although this work has contributed to making more testing methods available, the work presented here can serve as a basis for further research in future work. The following ideas show possible paths for the next projects:

#### **Authentication and Authorization:**

- **Token Expiry:** Test authentication mechanisms using expired or invalid tokens.
- **Role-Based Access:** Test access to endpoints with different roles to ensure proper authorization checks.
- **No Authentication:** Attempt to access protected endpoints without any authentication to verify access controls. Based on the information extracted from the schema, it is possible

to identify whether a particular endpoint requires authentication and thus generate requests validating this condition.

- Privilege Escalation: Check if lower-privileged users can perform actions reserved for higher-privileged users.

**Cross-Parameter Dependencies:**

- Parameter Combinations: Test various request parameter combinations to identify issues arising from specific parameter interactions.
- Conditional Logic: Check how the API handles conditional logic based on parameter values.

The methods described in this study have been evaluated using a targeted application. Subsequent research could involve validating this methodology in broader and more complex APIs.

# References

- [1] V. Atlidakis, P. Godefroid, and M. (2020) Polishchuk. Checking security properties of cloud service rest apis. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, page 387–397, 2020.
- [2] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: stateful rest api fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, page 748–758. IEEE Press, 2019.
- [3] Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, and Jordi Cabot. An openapi-based testing framework to monitor non-functional properties of rest apis. In Maria Bielikova, Tommi Mikkonen, and Cesare Pautasso, editors, *Web Engineering*, pages 533–537, Cham, 2020. Springer International Publishing.
- [4] C. Cheh and B. Chen. Analyzing openapi specifications for security design issues. In *2021 IEEE Secure Development Conference, SecDev*, page 15–22, 2021.
- [5] Roy T. Fielding, Mark Nottingham, and Julian Reschke. Rfc 9110: Http semantics, 2022. Available at <https://www.rfc-editor.org/rfc/rfc9110.html>, last accessed in June 2024.
- [6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000. Available at [https://ics.uci.edu/~fielding/pubs/dissertation/fielding\\_dissertation.pdf](https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf), last accessed in June 2024.
- [7] Patrice Godefroid. Fuzzing: hack, art, and science. *Commun. ACM*, 63(2):70–76, jan 2020.
- [8] Zac Hatfield-Dodds and Dmitry Dygalo. Deriving semantics-aware fuzzers from web api schemas, 2021.
- [9] HTTPWG HTTP Working Group. HTTPWG — Safe Methods, 2022. <https://httpwg.org/specs/rfc9110.html#safe.methods>.
- [10] Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. forest: A tree-based black-box fuzzing approach for restful apis. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 695–705, 2023.
- [11] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. A catalogue of inter-parameter dependencies in restful web apis. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings*, page 399–414, Berlin, Heidelberg, 2019. Springer-Verlag.

- [12] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Restest: automated black-box testing of restful web apis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 682–685, New York, NY, USA, 2021. Association for Computing Machinery.
- [13] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. Online testing of restful apis: promises and challenges. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ES-EC/FSE 2022, page 408–420, New York, NY, USA, 2022. Association for Computing Machinery.
- [14] Darrel Miller, Jeremy Whitlock, Marsh Gardiner, Mike Ralphson, Ron Ratovsky, and Uri Sarid. Openapi initiative, 2021. Available at <https://spec.openapis.org/oas/v3.1.0>, last accessed in June 2024.
- [15] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. Metamorphic testing of restful web apis. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2018.

## Appendix A

# List of Publications Included in the Literature Review

Table A.1: List of Publications

| Type             | Authors                                                  | Title                                                                    | Year | Publisher |
|------------------|----------------------------------------------------------|--------------------------------------------------------------------------|------|-----------|
| Conference Paper | Segura S, Parejo J, Troya J, Ruiz-Cortes A               | Metamorphic testing of RESTful Web APIs                                  | 2018 | IEEE      |
| Conference Paper | Martin-Lopez A, Segura SRuiz-Cortés A                    | A Catalogue of Inter-parameter Dependencies in RESTful Web APIs          | 2019 | Springer  |
| Conference Paper | Vaggelis Atlidakis; Patrice Godefroid; Marina Polishchuk | RESTler: Stateful REST API Fuzzing                                       | 2019 | IEEE      |
| Conference Paper | Theofanis Vassiliou-Gioles                               | A simple, lightweight framework for testing RESTful services with TTCN-3 | 2020 | IEEE      |
| Conference Paper | Vaggelis Atlidakis; Patrice Godefroid; Marina Polishchuk | Checking Security Properties of Cloud Service REST APIs                  | 2020 | IEEE      |
| Conference Paper | Phu X. Mai; Fabrizio Pastore; Arda Goknil; Lionel Briand | Metamorphic Security Testing for Web Systems                             | 2020 | IEEE      |

Continued on next page

**Table A.1 – continued from previous page**

| <b>Type</b>            | <b>Authors</b>                                                               | <b>Title</b>                                                                         | <b>Year</b> | <b>Publisher</b> |
|------------------------|------------------------------------------------------------------------------|--------------------------------------------------------------------------------------|-------------|------------------|
| Conference Paper       | Alberto Martin-Lopez, Sergio Segura, Antonio Ruiz-Cortés                     | RESTest: Black-Box Constraint Based Testing of RESTful Web APIs                      | 2020        | Springer         |
| Conference Paper       | Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, Jordi Cabot | An OpenAPI-Based Testing Framework to Monitor Non-functional Properties of REST APIs | 2020        | Springer         |
| Journals and Magazines | Sergio Segura; Dave Towey; Zhi Quan Zhou; Tsong Yueh Chen                    | Metamorphic Testing: Testing the Untestable                                          | 2020        | IEEE             |
| Conference Paper       | Steven Bucaille, Javier Luis Cánovas Izquierdo, Hamza Ed-Douibi, Jordi Cabot | An OpenAPI-Based Testing Framework to Monitor Non-functional Properties of REST APIs | 2020        | Springer         |
| Conference Paper       | Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk                      | Intelligent REST API data fuzzing                                                    | 2020        | ACM              |
| Journals and Magazines | Patrice Godefroid                                                            | Fuzzing: hack, art, and science                                                      | 2020        | ACM              |
| Conference Paper       | Ovidiu Baniş, Diana Florea, Robert Gyalai and Daniel-Ioan Curiac             | Automated specification-based testing of REST APIs                                   | 2021        | MDPI             |
| Conference Paper       | Carmen Cheh; Binbin Chen                                                     | Analyzing OpenAPI Specifications for Security Design Issues                          | 2021        | IEEE             |
| Conference Paper       | Martin-Lopez A, Segura S, Ruiz-Cortés A                                      | RESTest: automated black-box testing of RESTful web APIs                             | 2021        | ACM              |
| Conference Paper       | Zac Hatfield-Dodds; Dmitry Dygalo                                            | Deriving Semantics-Aware Fuzzers from Web API Schemas                                | 2022        | IEEE             |

Continued on next page

**Table A.1 – continued from previous page**

| <b>Type</b>            | <b>Authors</b>                                                                                          | <b>Title</b>                                                                      | <b>Year</b> | <b>Publisher</b> |
|------------------------|---------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------|------------------|
| Conference Paper       | Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso                                               | Automated test generation for REST APIs: no time to rest yet                      | 2022        | ACM              |
| Journals and Magazines | Ehsan A, Abuhaliqa M, Catal C, Mishra D                                                                 | RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions | 2022        | MDPI             |
| Conference Paper       | Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova                   | Type-directed program synthesis for RESTful APIs                                  | 2022        | ACM              |
| Conference Paper       | Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao | Morest: model-based RESTful API testing with execution feedback                   | 2022        | ACM              |
| Conference Paper       | Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés                                            | Online testing of RESTful APIs: promises and challenges                           | 2022        | ACM              |
| Conference Paper       | Felício D, Simão J, Datia N                                                                             | RapiTest: Continuous Black-Box Testing of RESTful Web APIs                        | 2023        | ScienceDirect    |
| Conference Paper       | Jiaxian Lin; Tianyu Li; Yang Chen; Guangsheng Wei; Jiadong Lin; Sen Zhang; Hui Xu                       | foREST: A Tree-based Black-box Fuzzing Approach for RESTful APIs                  | 2022        | IEEE             |
| Conference Paper       | Karishma Rahman; Clemente Izurieta                                                                      | An Approach to Testing Banking Software Using Metamorphic Relations               | 2020        | IEEE             |
| Journals and Magazines | Amid Golmohammadi, Man Zhang, and Andrea Arcuri                                                         | Testing RESTful APIs: A Survey                                                    | 2023        | ACM              |

## Appendix B

# RESTler and Application Files

### B.1 Dictionary

The dictionary file *dict.json* [B.1](#) is the file that RESTler builds based on types of data that could be identified in OAS and with default data that can be used during test executions. This file can be modified to add more values for fuzz mode.

### B.2 Spec Coverage

A portion of RESTler that records and reports testing coverage on the API specification is called *speccov.json* [27](#) file. It assists developers in recognising sections of an API exercised during its fuzzing, thus helping them identify untested or poorly tested sections.

### B.3 RESTler Sequence

The [Figure B.2](#) shows a graph created during the RESTler execution in *test mode*. This graph has been created from RESTler requests and responses, analysing requests by path and in a temporal way.

As we can see, requests are made following a sequence of dependencies between the types of HTTP methods, e.g., the `POST` method precedes a `GET`, and a `DELETE` follows a `POST` or `PUT`. This can be seen as a happy path laid out by the RESTler to cover all the operations available in the specification.

### B.4 BlogPost OAS

[Figure B.3](#) show the OpenAPI specification of the BlogPost application and the [Listing 28](#) show a fragment of the `yam` file.

```
{
  "restler_fuzzable_string": [
    "fuzzstring"
  ],
  "restler_fuzzable_string_unquoted": [],
  "restler_fuzzable_datetime": [
    "2019-06-26T20:20:39+00:00"
  ],
  "restler_fuzzable_datetime_unquoted": [],
  "restler_fuzzable_date": [
    "2019-06-26"
  ],
  "restler_fuzzable_date_unquoted": [],
  "restler_fuzzable_uuid4": [
    "566048da-ed19-4cd3-8e0a-b7e0e1ec4d72"
  ],
  "restler_fuzzable_uuid4_unquoted": [],
  "restler_fuzzable_int": [
    "10"
  ],
  "restler_fuzzable_number": [
    "1.23"
  ],
  "restler_fuzzable_bool": [
    "true"
  ],
  "restler_fuzzable_object": [
    "{ \"fuzz\": false }"
  ],
  "restler_custom_payload": {
    "body": [
      "first blog",
      "second blog"
    ]
  },
  "restler_custom_payload_unquoted": {
    "body": [
      "\"third blog\", \"id\":1"
    ]
  },
  "restler_custom_payload_uuid4_suffix": {
    "body": "body"
  },
  "restler_custom_payload_header": {},
  "restler_custom_payload_query": {}
}
```

Figure B.1: RESTler Dictionary

```

1  "1950cbddab7726489624c3d346d3426561c921ad_1": {
2      "verb": "GET",
3      "endpoint": "/api/blog/posts",
4      "verb_endpoint": "GET /api/blog/posts",
5      "valid": 1,
6      "matching_prefix": [],
7      "invalid_due_to_sequence_failure": 0,
8      "invalid_due_to_resource_failure": 0,
9      "invalid_due_to_parser_failure": 0,
10     "invalid_due_to_500": 0,
11     "status_code": "200",
12     "status_text": "OK",
13     "error_message": null,
14     "request_order": 0,
15     "sample_request": {
16         "request_sent_timestamp": null,
17         "response_received_timestamp": "2024-07-26 10:29:11",
18         "request_verb": "GET",
19         "request_uri": "/api/blog/posts?page=10&per_page=10",
20         "request_headers": [
21             "Accept: application/json",
22             "Host: localhost:8888"
23         ],
24         "request_body": null,
25         "response_status_code": "200",
26         "response_status_text": "OK",
27         "response_headers": [
28             "date: Fri, 26 Jul 2024 10:29:10 GMT",
29             "server: uvicorn",
30             "content-length: 46",
31             "content-type: application/json"
32         ],
33         "response_body":
34             ↪ "{ \"items\": [], \"per_page\": 10, \"page\": 10, \"total\": 0 }"
35     },
36     "tracked_parameters": {}
37 }

```

Listing 27: RESTler Specification Coverage Fragment

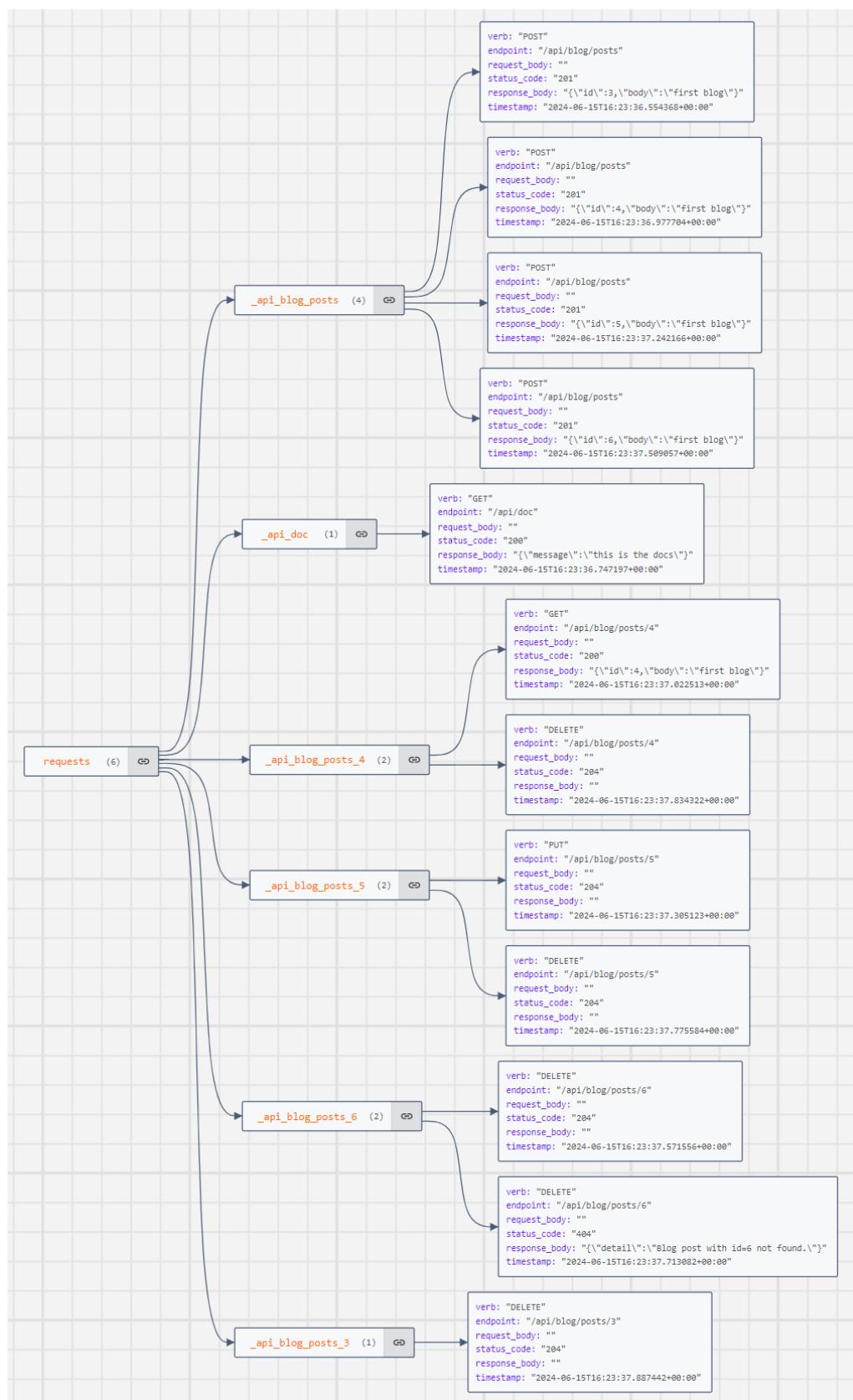


Figure B.2: Request Sequence

# Demo Blog Server

0.1.0 OAS 3.1

[/openapi.json](#)

Servers

http://localhost:8888 - Staging environment

## blog/posts

|        |                          |                  |   |
|--------|--------------------------|------------------|---|
| POST   | /api/blog/posts          | Create Post      | ✓ |
| GET    | /api/blog/posts          | Get Posts        | ✓ |
| PUT    | /api/blog/posts/{postId} | Update Blog Post | ✓ |
| GET    | /api/blog/posts/{postId} | Get Blog Post    | ✓ |
| DELETE | /api/blog/posts/{postId} | Delete Blog Post | ✓ |

## blog/posts/comments

|        |                                               |                          |   |
|--------|-----------------------------------------------|--------------------------|---|
| POST   | /api/blog/posts/{postId}/comments             | Create Comment           | ✓ |
| GET    | /api/blog/posts/{postId}/comments/{commentId} | Get Blog Post Comments   | ✓ |
| DELETE | /api/blog/posts/{postId}/comments/{commentId} | Delete Blog Post Comment | ✓ |

Figure B.3: BlogPost OAS

```

1  {
2    "openapi": "3.1.0",
3    "info": {
4      "title": "Demo Blog Server",
5      "version": "0.1.0"
6    },
7    "servers": [
8      {
9        "url": "http://localhost:8888",
10       "description": "Staging environment"
11      }
12    ],
13    "paths": {
14      "/api/blog/posts": {
15        "post": {
16          "tags": [
17            "blog/posts"
18          ],
19          "summary": "Create Post",
20          "operationId": "create_post_api_blog_posts_post",
21          "requestBody": {
22            "required": true,
23            "content": {
24              "application/json": {
25                "schema": {
26                  "$ref": "#/components/schemas/BlogPostPublicInput"
27                }
28              }
29            }
30          },
31          "responses": {
32            "201": {
33              "description": "Successful Response",
34              "content": {
35                "application/json": {
36                  "schema": {
37                    "$ref": "#/components/schemas/BlogPostPublicInput"
38                  }
39                }
40              }
41            },
42            "422": {
43              "description": "Validation Error",
44              ...
45            }
46          }
47        }
48      }
49      ...
50    }
51  }

```

Listing 28: OpenAPI schema for BlogPosts